

AD-A047 392

BOLT BERANEK AND NEWMAN INC CAMBRIDGE MASS

F/G 9/2

COMMUNICATIONS ORIENTED LANGUAGE (COL): LANGUAGE DEFINITION.(U)

MAY 77 A EVANS, C R MORGAN

DCA100-76-C-0051

UNCLASSIFIED

BBN-3534

SBIE-AD-E100007

NL

1 OF 4

ADA047392



AD-E 100 007

(3)  
DRC

R-3534

AD A047392

DEFENSE COMMUNICATIONS ENGINEERING CENTER

REPORT NO. 3534

COMMUNICATIONS ORIENTED LANGUAGE (COL):  
LANGUAGE DEFINITION

MAY 1977

DDC  
RECEIVED  
DEC 13 1977  
B

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

PREPARED FOR THE DEFENSE COMMUNICATIONS AGENCY  
BY BOLT BERANEK & NEWMAN INC.

AD No  
DDC FILE COPY



Report No. 3534

Bolt Beranek and Newman Inc.

COMMUNICATIONS ORIENTED LANGUAGE (COL):  
LANGUAGE DEFINITION

Arthur Evans, Jr.  
C. Robert Morgan

2 May 1977

Submitted to:

Mr. Paul M. Cohen, R-810  
Defense Communications Engineering Center  
1860 Wiehle Avenue  
Reston, Virginia 22090

This research was supported by the Defense Communications Agency  
of the Department of Defense, contract No. DCA 100-76-C-0051.

**DISTRIBUTION STATEMENT A**

Approved for public release;  
Distribution Unlimited

Reston, Virginia  
May 4, 1977

The Defense Communications Engineering Center is currently engaged in the development of a capability to efficiently and economically produce software for the Defense Communications System. The objectives of this effort are to provide management control for this software, to ensure that this software meets the requirements for the system factors such as speed, reliability and security, and to insure that such software is developed, tested, verified and maintained in a manner that is consistent with the evolving state-of-the-art.

This is a report covering work which has been performed by BBN in support of such efforts.

Comments on this report from all government, industry, and university sources are encouraged, and will be greatly appreciated. We look at such comments as a source of information for our future studies in this area. Please send comments to me at the address below or to my ARPANET address.

Paul M. Cohen, Code R810  
Defense Communications Engineering Center  
1860 Wiehle Avenue  
Reston, Virginia 22090

ARPANET Address: COHEN at BBN-TENEXB

ACCESSION for	
NTIS	Auto Section <input checked="" type="checkbox"/>
DDC	DDC Section <input type="checkbox"/>
STANDARDIZED	<input type="checkbox"/>
JUSTIFICATION	
BY	
DISTRIBUTION/AVAILABILITY CODES	
Dist. <input type="checkbox"/> and/or SPECIAL	
A	

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM	
1. REPORT NUMBER BBN Report No. - 3534	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER	
4. TITLE (and Subtitle) COMMUNICATIONS ORIENTED LANGUAGE (COL): LANGUAGE DEFINITION.	5. TYPE OF REPORT & PERIOD COVERED Final Report May 1976 - May 1977		
7. AUTHOR(s) Arthur/Evans, Jr. C. Robert Morgan	6. PERFORMING ORG. REPORT NUMBER BBN Report No. 3534		
9. PERFORMING ORGANIZATION NAME AND ADDRESS Bolt Beranek and Newman Inc. 50 Moulton Street Cambridge, MA 02138	8. CONTRACT OR GRANT NUMBER(s) DCA100-76-C-0051		
11. CONTROLLING OFFICE NAME AND ADDRESS Defense Communications Engineering Ctr. 1860 Wiehle Avenue Code R800 Reston, Virginia 22090	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS n/a		
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office) n/a	12. REPORT DATE 2 May 1977		
	13. NUMBER OF PAGES x + 321 335 p.		
	15. SECURITY CLASS. (of this report) unclassified		
16. DISTRIBUTION STATEMENT (of this Report) Approved for public release, distribution unlimited			
17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report) n/a			
18. SUPPLEMENTARY NOTES This report supersedes BBN Report No. 3261, "Development of a Communications Oriented Language, Part II of Final Report", 20 March 1976. BBN Report No. 3261, Part II is now obsolete.			
19. KEY WORDS (Continue on reverse side if necessary and identify by block number) Communications Oriented Language, COL, High Order Language, HOL, Language Evaluation, Language Design, Compiler Design, Communications Systems, Computer Communications, DOD1, Ironman.			
20. ABSTRACT (Continue on reverse side if necessary and identify by block number) A Communications Oriented Language (COL) is a high order language designed to be maximally effective as a vehicle with which to program communications applications. This document is a description of a design for a COL. The design philosophy and principles that lead to the COL are first discussed. Then the language is described, the syntax semi-formally using modified BNF and the semantics informally (cont. on other side)			

DD FORM 1473

EDITION OF 1 NOV 65 IS OBSOLETE

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

in English text. Some examples of COL programming are included. An appendix shows how machine-level instructions for a specific machine can be written in the COL, with an example showing interrupt processing.

This document is an extensive revision of BBN Report No. 3261, "Development of a Communications Oriented Language," 20 March 1976.

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)



### ABSTRACT

A Communications Oriented Language (COL) is a high order language designed to be maximally effective as a vehicle with which to program communications applications. This document is a description of a design for a COL. The design philosophy and principles that led to the COL are discussed first. Then the language is described, the syntax semi-formally using modified BNF and the semantics informally in English text. Some examples of COL programming are included. An appendix shows how machine-level instructions for a specific machine can be written in the COL, with an example showing interrupt processing.

This document is an extensive revision of BBN Report No. 3261, "Development of a Communications Oriented Language," 20 March 1976.

### ACKNOWLEDGMENTS

The language design reported here was performed by Arthur Evans, Jr., and C. Robert Morgan. During the early part of the design, important contributions were made by Professor Edgar T. Irons of Yale University and Walter D. Bilofsky of BBN. Later Joel B. Levin and William F. Mann each studied the document and wrote a program in the language, thereby providing important feedback to the design process. Recently the document was read carefully and commented on by Jerome R. Cherniack, Randall D. Rettberg and Richard R. Wolfson. It is our pleasure to acknowledge our gratitude to each of these people for their contributions to the design of the language and to the readability of this document.

We are particularly grateful to Paul M. Cohen of the Defense Communications Engineering Center of DCA, both for making technical contributions to the design of the language and for providing us an unhindered opportunity to perform this work.



## Table of Contents

Chapter 1	Introduction . . . . .	1
1.1	Background and Motivation . . . . .	3
1.1.1	Design Goals . . . . .	4
1.1.2	Design Philosophy . . . . .	9
1.1.3	Design Principles . . . . .	15
1.2	Summary of the Language . . . . .	16
1.3	Assumptions about the Implementation . . . . .	23
1.3.1	File System . . . . .	23
1.3.2	The Compiler . . . . .	24
1.3.3	The Linker . . . . .	27
1.3.4	Run Time Environment . . . . .	28
1.4	The "Tinman" and "Ironman" Proposals . . . . .	29
1.5	About This Document . . . . .	30
1.5.1	Organization of This Document . . . . .	31
1.5.2	Descriptive Method . . . . .	32
1.5.3	Conventions Used in this Document . . . . .	35
1.6	Character Set . . . . .	35
Chapter 2	Declarations . . . . .	36
2.1	Function and Routine Declarations . . . . .	40
2.2	Scalar Declarations . . . . .	48
2.2.1	Declaration of Variables . . . . .	49
2.2.1.1	Storage Class . . . . .	50
2.2.1.2	Type . . . . .	52
2.2.1.3	Initial Value . . . . .	53
2.2.2	Declaration of Constant Names . . . . .	53
2.2.3	Type Declaration . . . . .	56
2.3	Event Declarations . . . . .	59
2.4	Types . . . . .	60
2.4.1	Size . . . . .	61
2.4.2	Basic Types . . . . .	63
2.4.3	Aggregates . . . . .	65

## Table of Contents

2.4.4	Other Types . . . . .	73
2.5	Examples of Declarations . . . . .	77
2.6	Macro Declarations . . . . .	79
2.7	The Scope of Names . . . . .	81
2.8	Declaration Processing . . . . .	88
Chapter 3	Statements . . . . .	90
3.1	Simple Statements . . . . .	92
3.2	Conditional Statements . . . . .	94
3.3	Iteration Statements . . . . .	97
3.3.1	The Simple Iteration Statements . . . . .	99
3.3.2	The "for" statement . . . . .	101
3.4	Other Statements . . . . .	109
3.4.1	The "switchon" Command . . . . .	112
3.4.2	Zahn's Device . . . . .	113
3.4.3	Interlock Statements . . . . .	117
3.4.4	The "failure" Mechanism . . . . .	122
3.5	Restrictions . . . . .	124
Chapter 4	Expressions . . . . .	130
4.1	Primary Expressions . . . . .	131
4.2	Operator Expressions . . . . .	133
4.2.1	Precedence of the Operators . . . . .	135
4.2.2	Semantics of the Operators . . . . .	136
4.2.2.1	Semantics of Assignment . . . . .	138
4.2.2.2	Semantics of Parameter Passing . . . . .	140
4.2.2.3	The Arithmetic Operators . . . . .	141
4.2.2.4	The Relational Operators . . . . .	142
4.2.2.5	The Boolean Operators . . . . .	146
4.2.2.6	The Logical Operators . . . . .	146
4.2.2.7	The Shift Operators . . . . .	147

## Table of Contents

4.3	Aggregate Expressions . . . . .	147
4.4	Other Expressions . . . . .	153
4.5	Order of Evaluation . . . . .	157
Chapter 5	Lexical Matters . . . . .	163
5.1	Identifiers and Constants . . . . .	163
5.1.1	Identifiers . . . . .	163
5.1.2	Numeric Lexemes . . . . .	164
5.1.3	Character and String Constants . . . . .	166
5.2	Macros . . . . .	169
5.3	Compiler Directives . . . . .	171
5.3.1	Compiler Control . . . . .	171
5.3.2	Program Development . . . . .	172
5.3.3	Language Feature Control . . . . .	174
5.3.4	Miscellaneous Features . . . . .	174
5.4	Character Codes . . . . .	175
5.5	Miscellaneous Lexical Matters . . . . .	177
5.5.1	Spaces in the COL . . . . .	177
5.5.2	Comment Conventions . . . . .	178
5.5.3	Semicolon Insertion . . . . .	178
Chapter 6	Miscellaneous Topics . . . . .	182
6.1	Other Language Features . . . . .	182
6.1.1	Flexible Arrays and Variadic Procedures . . . . .	182
6.1.1.1	Introduction to the Concepts . . . . .	183
6.1.1.2	Flexible Arrays . . . . .	188
6.1.1.3	Variadic Procedures . . . . .	190
6.1.2	Macros . . . . .	190
6.1.2.1	Abbreviations for Types . . . . .	191
6.1.2.2	Open Subroutines . . . . .	193
6.1.2.3	Conventional Macros . . . . .	194
6.1.3	Encapsulated Types . . . . .	197
6.1.4	Exception Handling . . . . .	202
6.1.4.1	The "event" Declaration . . . . .	202
6.1.4.2	Zahn's Device . . . . .	206
6.1.4.3	The "failure" Mechanism . . . . .	208
6.1.5	Coercions . . . . .	210

## Table of Contents

6.1.6	Conversion Rules . . . . .	213
6.1.7	Defaults . . . . .	215
6.1.8	Environment Queries . . . . .	216
6.2	Issues Relating to Separate Compilation . .	217
6.2.1	Introduction . . . . .	217
6.2.2	Syntax for Separate Compilation . . . .	222
6.2.3	Notes on the Implementation . . . . .	224
6.3	Machine Dependent Features . . . . .	225
6.3.1	Machine-Like Code . . . . .	226
6.3.2	Data Layout in Memory . . . . .	227
6.3.3	Absolute Addressing . . . . .	229
6.4	Assertions . . . . .	230
6.5	Run-Time Library . . . . .	231
Chapter 7	Examples . . . . .	237
7.1	Simple Examples . . . . .	237
7.2	Formatted Output: a Variadic Procedure . .	239
7.3	Interlocks and Zahn's Device . . . . .	242
7.4	Binary Trees . . . . .	243
7.5	An Encapsulated Type . . . . .	248
7.6	Quick Sort . . . . .	252
References	Bibliographic References . . . . .	254
Appendix I	Summary of Syntax . . . . .	257
Ap1.1	Syntactic Classes . . . . .	257
Ap1.2	Listing of the COL's Complete Syntax . . .	260

## Table of Contents

Appendix II	Machine-Level Code for the SUE . . . . .	271
Ap2.1	Description of the SUE . . . . .	271
Ap2.1.1	Changes to the COL for the SUE . . . . .	273
Ap2.1.2	Registers in the SUE . . . . .	274
Ap2.1.3	Addressing Modes . . . . .	275
Ap2.1.4	SUE Instructions . . . . .	279
Ap2.1.5	Events in SUE . . . . .	283
Ap2.1.6	Interrupts and Input/Output . . . . .	284
Ap2.1.7	Unsigned Integer Multiply . . . . .	287
Ap2.2	Parallel Processing and Co-Routines . . . . .	289
Ap2.2.1	Functions for Parallel Processing . . . . .	289
Ap2.2.2	Implementation of START and JOIN . . . . .	291
Ap2.2.3	Co-routines . . . . .	295
Ap2.2.4	Interrupt Handling . . . . .	299
Appendix III	Terminology Used in this Document . . . . .	302
Appendix IV	Comparison with the "Ironman" . . . . .	311
Appendix V	Proposed Changes in the COL . . . . .	317
Index	Index . . . . .	320



## Figures

1-1	Example: Matrix Multiply	19
1-2	Example: Interchange Sort	20
1-3	CRC Checksum	21
2-1	Adding Apples and Oranges	58
2-2	Example of a Structure Declaration	69
2-3	Host to Host Message Format in ARPANET	71
2-4	Scope Rule Illustrated	83
2-5	Scope in a Procedure Body	85
2-6	Declaring Mutually Recursive Procedures	87
2-7	Use of "undeclare"	88
3-1	Sample Conditional Statement	97
3-2	Semantics of "for defined step until"	103
3-3	Semantics of "for step until"	103
3-4	Semantics of "for defined incr"	104
3-5	Semantics of "for incr"	105
3-6	Semantics of "for defined decr"	105
3-7	Semantics of "for decr"	106
3-8	Semantics of "for to"	106
3-9	Semantics of "for defined to"	107
3-10	Semantics of "for in"	108
3-11	Semantics of "for defined in"	109
3-12	Example of Zahn's Device	114
3-13	Equivalence for Zahn's Device	116
3-14	The "failure" Mechanism	123
4-1	Examples of COL Precedence	137
4-2	Example of Aggregate Usage	152
6-1	The Variadic Function MaxI	187
6-2	Binding of Free Variables in Types	192
6-3	Scope in Open Procedures	194
6-4	Example of Macro	196
6-5	An Encapsulated Data Type	199
6-6	Example of the "overflow" Event	204
6-7	Zahn's Device and "overflow"	207
6-8	Capturing a "fail" for Finalization	211
6-9	Coercion of "different" Types	212
6-10	Example of Separate Compilation	221

## Figures

6-11	Sample Structure . . . . .	232
6-12	The Module FREE_PACKAGE . . . . .	235
7-1	PRINT: a Variadic Procedure . . . . .	240
7-2	Example: Zahn's Device . . . . .	244
7-3	Example: Tree Walking . . . . .	246
7-4	Example: Routine for Walk . . . . .	247
7-5	Example: Use of Walk . . . . .	247
7-6	QUEUE: An Example of a Capsule . . . . .	249
7-7	Quick Sort Routine . . . . .	253
Ap2-1	Declarations for SUE Registers . . . . .	275
Ap2-2	Bits of the Status Register in SUE . . . . .	275
Ap2-3	Declaration for Status Bits . . . . .	276
Ap2-4	Declaration of Sample Variables . . . . .	277
Ap2-5	Addressable COL Expressions . . . . .	278
Ap2-6	COL Code for CMP and TST . . . . .	282
Ap2-7	Teletype Control Block . . . . .	286
Ap2-8	Code for Teletype Interrupt . . . . .	287
Ap2-9	Teletype Interrupt Code . . . . .	288
Ap2-10	Integer Multiply . . . . .	288
Ap2-11	SEND and GET Procedures . . . . .	291
Ap2-12	Variables for Process Switching . . . . .	293
Ap2-13	Code for START . . . . .	294
Ap2-14	SENDER and GETTER for Co-routines . . . . .	296

## Tables

3-1	Language Restrictions . . . . .	127
4-1	Infix and Prefix Operators . . . . .	134
4-2	Relative Precedence of COL's Operators . . . . .	136
4-3	Permitted Operands for Operators . . . . .	137
4-4	COL's Relational Operators . . . . .	143
4-5	The boolean (and logical) Operators . . . . .	146
4-6	The functions truncate, round, floor, ceiling . . . . .	155
5-1	Character Code * Conventions . . . . .	168
5-2	ASCII Character Codes in the COL . . . . .	176
5-3	Lexeme Pairs Violating the Semicolon Rule . . . . .	181
7-1	Examples of COL Usage . . . . .	238

## Chapter 1: Introduction

The programming of computers to perform communications applications involves much expenditure of effort within the Department of Defense. In the past, all such programming has been done in assembly language, with high attendant costs in both programmer productivity and difficulty of maintenance. In an attempt to reduce these costs, the Defense Communications Agency (DCA) has requested that Bolt Beranek and Newman Inc. (BBN) develop a high order language (HOL), oriented towards communications applications, to do this programming. In an earlier contract with DCA (contract number 100-DCA-75-C-0051), BBN studied the needs of communications programming and investigated existing languages to determine the extent to which they met these needs. The results of this study are reported in BBN Report No. 3261, "Development of a Communications Oriented Language", 20 March 1976. The conclusion reached was that no single language met adequately the perceived needs of programming communications applications. A Communications Oriented Language (COL) was therefore designed to meet the specific needs of such programming.

In the present contract, for which this document is part of the final report, BBN has refined the design of the COL, on the basis of further language study, of a preliminary compiler design

for the COL, and of inputs from communications programmers not part of the COL design team. This report is the result of the language study -- it is an extensively revised and expanded version of Report No. 3261. A companion volume [Morgan-77], which constitutes the other part of the final report, describes a proposed compiler for the COL.

The bulk of this document describes the COL language, using a modified version of BNF to present the syntax and informal discussion in English prose to describe the semantics. The present chapter, which outlines the philosophy of programming language design that has governed our deliberations, gives some of our reasons for choosing the design we have. The final chapter provides some examples of programming in the COL, and additional matter is relegated to appendices. The reader who wants merely to learn the language and is not interested in its motivation may skip the rest of this chapter, except for section 1.2, which is an overview of the COL language, and section 1.5, which provides information about this document that the reader requires.

Computer language design often involves a large measure of creative plagiarism. We have used good ideas wherever we found them, and the discerning reader will note much that is familiar. Clearly, BCPL and PASCAL (including in particular its EUCLID dialect) have had a significant impact on our thinking, as have



also PL/I, JOVIAL, CS-4, ALPHARD and BLISS(1). In addition, we have been influenced by the criteria advanced by the Department of Defense's High Order Language Working Group (HOLWG), as expressed in the so-called Tinman Report [Fisher-76]. It is our pleasure to acknowledge our debt to practically everyone we know in the profession.

### 1.1: Background and Motivation

The COL, as its name indicates, has been designed to be oriented towards communications applications. Much design effort during the first contract went into the question of what linguistic facilities might be appropriate to communications applications. At first it appeared to us that the needs of a system programming oriented language (SPOL) and of a COL were similar; further consideration revealed essentially no significant differences. Programming communications applications imposes stringent requirements in such areas as efficiency of object code, direct access to machine facilities, efficient bit manipulations, manipulating complex data structures, producing very large programs by a team of programmers, maintaining and upgrading the program after the original team has disbanded, etc. These requirements also apply to system programming, and any candidate for a SPOL must necessarily meet them. This conclusion in no way

---

(1) Bibliographic citations for computer languages referred to in the text appear in a reference section after the last chapter.

detracts from the value of the COL for its intended applications -- it is merely interesting.

A second important question has to do with what specific communications facilities should be built into the COL. Communications applications deal with data structures such as stacks and queues, with program structures such as interrupts and co-routines, with parallel processing, and with other sophisticated concepts from the repertory of the skilled programmer. What facilities should be part of the COL to deal with these? We gave the matter much long and careful thought, coming finally to the conclusion that none of them should be built into the COL. Instead, the COL has been designed with adequate flexibility and generality that the programmer can readily implement them to meet his immediate needs. This point is addressed at length in section 1.1.2, in which the thinking that led to this conclusion is presented.

The remaining subsections of this section discuss our design goals, philosophy, and principles. Study of these topics should reveal much about why the COL turned out as it did.

#### 1.1.1: Design Goals

The design goals that led us to this design for a COL are of course based on the requirement that the COL be a useful tool for the programmer involved in a communications application. Consistent with that requirement, then, the goals we have tried to

attain may be stated as follows:

- . Efficiency of the compiled code is of paramount
- . Portability is needed, that is, the ability to move a program from one machine to another.
- . Completeness is required, in the sense that it must be possible to program all of any application in the COL without recourse to use of an assembler.
- . Readability of COL programs is necessary. In particular, it is of greater importance than is writability.
- . Large projects will be programmed in the COL, imposing requirements involving communication between programmers and communication between separately compiled modules.
- . Highly skilled programmers will be the users of the language, so that ease of learning is less important than is ease of use.

The first two points and the interactions between them are now addressed in detail, followed by briefer discussions of the remaining points.

The primary requirements for communications applications are efficiency of the compiled code and portability, in that order.

We feel that the sine qua non of COL design is that it be possible, at least locally, to produce programs that make maximally effective use of the hardware resources. Portability runs a close second in importance, but it is not possible to optimize both simultaneously. In all cases where we were forced to choose between these two goals, we opted to maximize efficiency. Where possible we have sought portability; where it had to be sacrificed we selected features that make the machine-dependent parts of programs as conspicuous as possible. Thus we require that inclusion of a machine-dependent feature be preceded by a warning flag that alerts both the compiler and the most casual reader of the program. For example, the programmer can override the language's type-checking mechanism, but it is easy to see when this is being done. Contrast the FORTRAN approach, in which this is done by equivalencing an integer to a real, with the equivalence statement probably in a remote part of the program text.

The requirement for efficiency has had one other impact on the design. It is very easy in designing a language to provide facilities which, while perhaps desirable from the programmer's point of view, may have serious implications on efficiency, sometimes making it hard to compile good code and sometimes making it well nigh impossible. We have thought through the implications of each proposed feature on run time efficiency. No matter how desirable a particular feature might be in the COL, we have



rejected it if we could not design a reasonably efficient implementation. For example, the literature contains many suggestions for exception handling, some of which are extremely powerful. However, our study has revealed that many of them impose run time inefficiencies that cannot be tolerated in a communications environment. We have thus proposed a mechanism which, while not so powerful, is capable of being implemented efficiently. In particular, presence of the feature in the language imposes no cost at all on sections of code that do not use it.

At least in selected parts of his program, the programmer must be able to achieve the most efficient possible code. In the extreme case, he must be able to specify an exact sequence of machine instructions. To this end, the COL includes a facility for machine-level code intermixed with commands written in COL. This facility is presented in section 6.3, and Appendix II contains a worked out example of its application on a specific computer. It is a design criterion of the COL that the language be adequately powerful in this area that there be no need for any assembler coding. That is, all code resident in a communications computer can be expressed in the COL. (Obviously, some of that code is not at all portable.)

We have kept in mind the fact that maintenance of programs is a significant part of the cost of large programming projects,



often exceeding over the lifetime of the project the cost of the development. We have regarded ease of reading a COL program as being more important than ease of writing, since it is usually the case that a group other than the original programming team is responsible for maintenance and later changes.

In a large programming effort, the program is broken into segments that are compiled separately. The COL provides features that facilitate such a mode of use, with effective and safe methods of communicating between separately compiled segments.

The nature of the applications for which the COL is intended is such that highly skilled programmers will do the work. Thus ease of learning the language by novices is not a requirement. Stated differently, the language is learned (by a programmer) only once but will be used for many years. Ease of use is more important than ease of learning.

The COL will be used in the construction of large programs produced by many programmers working together. In projects of this size, good management techniques are essential for success. Although development of management techniques has not been part of our charter, we have attempted in our design to provide those tools which we feel are required for good management. As noted previously, the COL has been designed so that machine dependencies are conspicuously flagged. A COL compiler could therefore create from each compilation a separate report listing the machine

dependencies used, and a management decision could then cause that report to be sent to an appropriate manager. Such decisions are not part of the COL design; what we have attempted is to make it possible to implement such decisions where appropriate.

#### 1.1.2: Design Philosophy

Given that the COL is to be used by highly skilled programmers to carry out an unusually demanding programming task, we have tried to provide in it those tools which we felt were needed. Without being fanatics on the subject, we are believers in that discipline that has come to be referred to as "structured programming". We have, for example, provided the user with a sufficiently rich selection of control structures that he should seldom feel the need to write a "goto". On the other hand, we are familiar enough with the exacting demands and specialized requirements of communications programming that we have provided a restricted "goto" statement for those cases where it is the best tool to accomplish the task. The field of structured programming has supplied a rich literature on which we have tried to capitalize, selecting those features that seem best suited to our needs. Data structuring is less well understood; nonetheless, we have tried to provide what seem to us to be useful facilities.

Early in the design effort, after giving long and serious consideration to the issue of data types, we decided that the COL is to be a fully typed language. This decision means that every

variable must be fully declared before it may be used, and it also implies that every use is checked before execution to insure that it matches the declaration. The two key points to the previous sentence are that EVERY use is checked, and that the checks are performed BEFORE EXECUTION. For example, parameters supplied to procedures must be checked against the declared argument types. This check is performed even if the call and the procedure declaration are compiled separately. Consistent with the requirement for efficiency (in both space and time) of running COL programs, all such checking is performed before the program is executed. (Most of the checking is done at compile time, but some final checking is required as part of the linking process.)

Although the COL is fully typed, methods are available that permit the programmer to circumvent this type-checking; the language is designed so that such circumvention is immediately obvious to the reader.

The communications environment differs in important ways from machine to machine, and the tools required vary depending on the machine. We have therefore elected to leave out of the COL those facilities that can better be programmed as subroutines or macros, written (presumably) using the machine-dependent facilities of the COL. The COL therefore has built into it no facilities for input/output, no stacks or queues, no string operators, no specific mechanisms for parallel processing (other than

interlocks), no co-routine syntax. All of these features can be implemented by programs written in COL, and BBN's extensive experience in programming communications applications on many different hardware architectures suggests that this is the correct decision.

Consider, for example, input/output. This term refers in communications applications to two different activities that must be distinguished: the actual communication function itself, in which bits are sent and received over communication channels, and the preparation of formatted data, such as reports to the operator. The former is invariably both highly critical to the overall bandwidth of the application and also highly dependent on the details of the particular computer. Although appropriate macros can readily be provided for any given situation, we were able to define no specific linguistic tools of wide application. We have therefore put into the COL nothing specific to this function, because we feel that the details are too diverse and require efficient implementation. Instead, we have created a language in which the required specialized communications functions can be easily implemented in an efficient and readable form. As our review of existing languages showed, no existing language can perform these functions to the extent needed and possible. Formatting messages for human consumption, the other type of input/output, is rarely an important matter in communications applications. Inasmuch as it can easily be



performed by subroutine, we have omitted it from the COL. (Section 7.2 shows as an example of COL programming a procedure appropriate for formatted output.)

The COL contains no linguistic constructs specifically adapted to queues or stacks. Our observation of communications programming suggests no way to implement them that is of wide application. One Pluribus communications program that we are familiar with uses four radically different ways to implement queues. It therefore seemed more appropriate to provide general purpose tools for such implementations than to select some particular technique and build it into the COL. One such technique for queues, designed to be safe from external tampering, is shown in section 7.5.

There are in the COL no specific operators for manipulating character strings. Character strings are represented as packed arrays of characters, a representation that permits many operations on them to be performed directly and the rest by easily programmed subroutines. Such character string processing (concatenation, substring, etc.) is not an important part of communications programming.

The COL includes a run time free storage package. An "allocate" function returns a pointer to space in it, and a "free" statement may be used to return such space when it is no longer required. No automatic garbage collection mechanism is provided,

nor is there any automatic facility for returning such space: it is the programmer's responsibility to insure that space is returned when no longer needed.

The COL provides a mechanism that permits the programmer to include any desired sequence of machine instructions in the compiled program. It was clear to us early in the development of the COL that the stringent demands of communication programming require this effect, and much thought was given to various ways to achieve it. We rejected the idea of using assembly coding as such, opting instead for providing facilities that permit the programmer to specify arbitrary instruction sequences in a notation that is essentially the COL. Within code brackets (i.e., between the reserved words "code" and "endcode") the programmer has access to special built-in functions and routines that correspond to specific machine instructions. With appropriate documentation, a programmer can read such code knowing only the COL and the semantics of these procedures, which can be expressed in COL. It is this ready ability to read machine-like code that was crucial in our decision to use this technique. It is consistent with our design goal that readability is more important than writability. To help give credence to this idea, we have included as Appendix II a description of the extensions needed in the COL to program for a particular computer -- the Lockheed SUE. We are aware that this is not a common way to include machine-level code in a high level language, although it has been

used with success in the language IMP, but we have concluded after extensive study that it is the best way for our purposes.

The COL uses reserved words. This means that tokens in the language such as "declare", "integer", "if", etc., may not be used by the programmer as variables. Although this restriction requires that each programmer be aware of all of the reserved words of the language, we feel this is a small cost for the expert programmer for whom the COL is intended. This is a decision that improves readability at a small cost in ease of writability.

The COL distinguishes statements from expressions, not permitting a statement in an expression context. (A statement is a fragment of code such as an assignment statement or an iteration statement that is obeyed for its effect, while an expression is a fragment such as a sum or a function invocation that is evaluated to determine its value.) We feel that to do otherwise encourages an obscure coding style that is hard to debug and hard to read.

We have assumed that the COL will be implemented using a stack at run time, since we feel that the advantages of this approach outweigh the costs. However, as we are aware that some small communications computers cannot gracefully support a stack, we have insured that only minor changes are needed in the COL if there is no stack. Reentrant code (including recursion) would of course no longer be supported, and a different implementation of automatic storage would be required. Such an implementation would

be somewhat more efficient in use of space than is the usual Fortran method, since parallel blocks could share storage, although less efficient than a stack.

### 1.1.3: Design Principles

There is no recognized set of principles of programming language design, but we have attempted to govern our thinking by such principles as we have been able to formulate. We now list a few of them:

Ease of verification of programs written in the COL is important. If use of a particular language feature in a program causes the program to be hard to verify, then that feature is suspect and should be re-examined carefully. To check that we have met this criterion, we have developed proof rules for certain parts of the COL. (These are described in [Morgan-77].)

The principle of minimum astonishment is important. We feel that there is a flaw in a language design if the programmer is astonished to learn about some aspect of the language. We have tried to design a language that will seem sensible to both the average skilled programmer and the language specialist.

There is a principle of economy of writing. The programmer should have to write a lot to cause the compiler to compile expensive code. This means, for example, that simple-looking expressions should not compile into expensive subroutine calls.



An example of the violation of this principle is the varying string in many implementations of PL/I, in which any use of a varying string in a block can produce high overhead not just at the point of use but at block exit. The penalty can be quite high, since the compiler must insure that the block exit code is obeyed even if the block is exited in a non-standard way. In the COL, this design principle lead us to omit string operators from the language.

If a language feature is hard to explain, or if the explanation seems forced and awkward, or if we have felt a bit embarrassed about explaining a feature, then we have carefully re-examined it. If it's hard to explain, it's suspect.

## 1.2: Summary of the Language

It is difficult in a manual such as this to determine in what order to present the elements of the language. No matter what is described first, the authors (and the reader) find that other matters need to have been presented earlier. The choice made in this document is to present declarations first, followed by statements and then expressions. To assist the reader, this section presents an overview of most of the language.. An important part of its purpose is to show the reader how much of the COL resembles language features with which he is already familiar. We proceed mostly by example.

The fragment of COL text shown below multiplies together two N by N matrices A and B, storing the product into C.

```
    for I := 1 to N do
      for J := 1 to N do
        T := 0.0
        for K := 1 to N do
          T := T + A[I, K] * B[K, J]
        endfor
        C[I, J] := T
      endfor
    endfor
```

This simple example illustrates several points. Although the COL does not distinguish (except in character strings) between upper- and lower-case letters, the examples in this manual consistently use lower-case letters for reserved words (such as "for", "to", "do", etc.), and upper-case letters for identifiers in the program (such as "I", "J", "A", "T"). The purpose is to assist the reader still learning the language. Note that each "for" statement has a matching "endfor", the body being all statements between "do" and "endfor". As a typographic convention, the matching end is lined up vertically below the opening "for", with all intervening lines (the body) indented by four spaces. Again, this is not part of the COL language but has been found to make COL code easier to read.

The above example makes sense only within the scope of declarations for the variables used. For example,

```
declare(T: float)
```

might have been used to declare the temporary "T" used to accumulate the sum. The iteration variables "I", "J" and "K" need

not be declared, since a "for" statement has the effect of declaring its iteration variable. The arrays might have been declared by

```
declare(A, B, C: array[1..N], [1..N] of float)
```

However, the COL requires (except in rather special cases) that the limits of arrays be known at compile time. Although all occurrences of "N" above could be replaced by an integer (such as 10), another possibility is to define N to be a constant known at compile time, like this:

```
declare(N = 10)
```

The effect of such a declaration is as if each occurrence of "N" were replaced by the value of the expression to the right of the "=". That expression may be any expression that can be evaluated at compile time and is evaluated at the time the declaration of "N" is encountered. The COL requires that items be declared before use, so the order of appearance in the above text would not be correct. All this code is collected in an acceptable order in Figure 1-1. Note that all three declarations above are collected into a single declaration in the figure. Note also the "..." in the figure to indicate missing code not part of the example. Presumably that code stores values into matrices A and B. Although this is not the sort of program likely to be written in a language designed for communications applications, it was chosen to show that much in the COL is quite ordinary.

```
declare
(  N = 10
  A, B, C: array [1..N], [1..N] of float
  T: float
)

...

for I := 1 to N do
  for J := 1 to N do
    T := 0
    for K := 1 to N do
      T := T + A[I, K] * B[K, J]
    endfor
    C[I, J] := T
  endfor
endfor
```

Figure 1-1: Example: Matrix Multiply

-----

As a second example we look at a simple exchange sort routine, shown in Figure 1-2. The figure illustrates both of the COL's comment conventions. The mark "//" and all text to its right on the line are treated as a comment, and "/\*" introduces a comment that extends through the following "\*/". The former comment convention is used in this example to annotate the text and the latter is used (only in this section) to provide line numbers for reference. The method used in this program is to pass through the array interchanging successive elements if they are not in ascending order, terminating when a pass finds no elements out of order. (This is a COL manual, not an introduction to



```

/* 1*/ declare (N = 100)
...
/* 2*/ declare (V: array[1..N] of integer)

/* 3*/ // Store some values into V.
...

/* 4*/ // Sort array V into ascending order.
declare (OK: boolean)

/* 5*/ repeat
/* 6*/     OK := true // Nothing out of order yet found.
/* 7*/     for K := 2 to N do
/* 8*/         if V[K-1] > V[K] do // Elements out of order.
/* 9*/             swap(V[K-1], V[K]) // Interchange them.
/*10*/             OK := false // Something was done.
/*11*/         endif
/*12*/     endfor
/*13*/ until OK
// The array is sorted.

```

Figure 1-2: Example: Interchange Sort

-----

efficient sorting methods.) The boolean OK is set to "true" before each pass and set to "false" whenever a swap is done. Lines 5 and 13 and all intervening text are a "repeat" statement, of the form

repeat SS until E

(where SS here stands for one or more statements). The semantics is that SS is executed repeatedly until at the end of such an execution the expression E is found to be true. The "swap" statement on line 9 is an action built into the COL; it could of

course be replaced by three assignment statements. Note that the indentation rules enunciated above have been followed here.

More related to communications applications is the following fragment that performs a CRC checksum. The code is in Figure 1-3.

```
-----  
  
// CRC Checksum Routine  
  
/*1*/ declare (CHECKSUM: static 16 bit logical initially 16#0)  
  
/*2*/ routine ADD_BYTE(C: char)  
/*3*/   declare (SHIFT_CHAR: 8 bit logical)  
/*4*/   SHIFT_CHAR := force(8 bit logical: C)  
/*5*/   for I := 1 to 8 do  
/*6*/     CHECKSUM := (CHECKSUM rshift 1) xor  
                   when (CHECKSUM xor SHIFT_CHAR) = 0  
                     then 16#A001 else 16#0  
/*7*/     SHIFT_CHAR := rshift 1  
/*8*/   endfor  
/*9*/ endroutine
```

Figure 1-3: CRC Checksum

-----

In line 1 a static variable CHECKSUM is declared and initialized. Since it is a variable of type logical, its initial value must have that type. In the COL a logical constant is written as a base followed by the mark "#" followed by an integer. Note the base 16 constant on line 6, for example. The rest of the example

is a routine ADD\_BYTE. (The character underscore "\_" may be used in names to improve readability.) Its argument is a character C which is accumulated into the checksum already stored into CHECKSUM, so each invocation of ADD\_BYTE adds in another character. The body of ADD\_BYTE starts on line 3 with declaration of a local variable SHIFT\_CHAR which is an 8 bit logical. The assignment statement on line 4 causes the bit pattern that was in the parameter C to be stored into SHIFT\_CHAR. The COL requires that the type of the expression on the right match that of the variable on the left, so writing

```
SHIFT_CHAR := C
```

would result in an error diagnostic at compile time. The expression involving "force" causes C's bit pattern to be interpreted by the compiler as if it were an 8 bit logical, thereby defeating the COL's strong type checking. This routine has characteristics that are implementation dependent, the 16 bit logical. The operator "\*" in line 7 provides a simple way to write a construction that is used frequently in programming. That line could equivalently have been written

```
SHIFT_CHAR := SHIFT_CHAR rshift 1
```

That is, in the COL a statement of the form

```
E1 *= op E2
```

is treated as if the programmer had written

```
E1 := E1 op E2
```

where "op" is any of the COL's infix operators.

### 1.3: Assumptions about the Implementation

This document describes a programming language and not an operating system. Nonetheless, the language as described places a few requirements on the implementation and the environment in which it resides.

#### 1.3.1: File System

We have assumed that the COL programmer submits his programs to the compiler in an environment that includes a "file system". This means that there is resident in the computer a collection of named entities to which the programmer may refer. Whether this facility is called a pool or a library or some other name is irrelevant; what counts is that entities can be accessed from it by name. The file system is used in two ways. A fragment of COL text submitted alone to the compiler is called a MODULE, and each module has a name. Part of the text of a module may direct the compiler to examine the text of some other module, the other module being identified by its name. The compiler must be able to find such text given a module name. For example, in TENEX the text for a module named BILL might be in a file named BILL.COL.

The programmer may include in his program a directive of the form

```
% include DECLS.COL ;
```

to direct the compiler to replace this directive by the file named



"DECLS.COL"(2). Such a file may contain a single copy of information that must be included in several modules. It is good practice that there exist in the computer only one copy of important data, since otherwise there is no safe way to insure that the multiple copies are all changed at the same time.

There is one additional requirement placed on the file system. The compiler must be able to tell of a given file when it was created. Here "when" refers to calendar date and time.

#### 1.3.2: The Compiler

The design of the COL involves a few assumptions about characteristics of the compiler beyond the obvious ones that are linguistic in nature. The most important of these is that the compiler produce code of extremely high quality, code which approaches or exceeds the quality of code produced by a skilled assembly language programmer(3). Any COL implementation must start with the assumption that the compiler is large and expensive.

---

(2) The syntax of file names is not part of the COL but is dependent on the implementation. This is just a possible example -- it might be used on TENEX.

(3) This is not an unreasonable assumption. The BLISS compiler for the PDP-11 produces better code than do most programmers. Although a highly skilled programmer can do better than the compiler by making a special effort, the programmer is unlikely to maintain that effort in all his work. In particular, he is quite unlikely to expend that effort in modifying or repairing a program, so the final product will probably not be so finely tuned. The compiler never has off days.

There are additional requirements. We expect the compiler to produce clear and meaningful error diagnostics, to complete the compilation and produce binary output even in the event of errors detected in the input, and in general to be a high quality compiler consistent with the forefront of the state of the compiler building art. These are requirements of any good compiler, but there exist certain additional requirements that are more specific to the COL.

An important product of each compilation is a complete annotated listing file that shows the source code (with line numbers) along with compiler comments. The following points are illustrative:

- . All defaults supplied by the compiler are listed.
- . All machine-dependent code is flagged. This includes code in "code" brackets (section 6.3), all uses of variables declared to reside in a register or in a specified memory location, all uses of "force" to override the compiler's type-checking, all uses of the type "general", etc. The flagging is immediately obvious to the casual reader.
- . A concordance of identifier usage is available, keyed to line numbers in the listing.
- . Error messages are placed close to the offending part of the program text.

The compiler produces warning messages concerning potential problem areas that it can detect. However, for each such warning there is a construct in the language that the programmer can use to suppress the warning. That is, if the programmer says in effect, "I really mean this," then he does not get warned. It is our observation that if a system produces too many warning messages, most programmers eventually ignore them all. The COL compiler must not fall into that trap.

The output of the COL compiler includes a relocatable binary file ready for the linker and the annotated listing of the source program just described. Optionally, the compiler also produces a so-called assembly listing, showing the machine instructions it compiles, in a format similar to that produced by an assembler. To the extent practical, the original program appears among the instructions as comments. (This is not very practical in a highly optimizing compiler such as the COL's.) It is important to note that the compiler does NOT produce a file suitable for input to an assembler, for two reasons. First, use of such a file is almost guaranteed to conflict with good management principles, inasmuch as there is no practical way to enforce updating the COL source file if changes are made to the compiler's output. It is our intent in designing the COL that the provisions for insertion of machine-level code be sufficient that such post-editing not be required. Second, it is extremely difficult for a programmer to make changes correctly in the output of a high quality optimizing

compiler.

In many cases the COL compiler will be a cross compiler, running on a large machine to produce code for a small one. Many computers used in communications applications are too small to support a compiler of the sophistication required.

### 1.3.3: The Linker

The specification of the COL has been made on the assumption that creation of a running program takes place in two phases: First all modules of the program are compiled, and then a binding operation takes place. This latter operation, referred to hereafter as linking, resolves linkages between the separately compiled pieces. Although linkers are well known and understood, the COL requires that the linker perform certain additional functions. One is to make the checks having to do with the strong data typing mentioned earlier. Two aspects of the language require this checking: procedure arguments and references to external data. In each case the source program includes all of the type information necessary, and the compiler checks that the code matches the declared types. The linker must insure that the same information is given to the various modules. Note that the linker does not change the compiled code -- it merely checks for consistency. (Another way to think of this process is that the compiler inserts into the binary output assertions that match what the programmer said to be the types of objects passed between

modules, and the linker checks that these assertions are consistent between modules.) Note also that ALL of the checking is performed before execution of the program. Further, this type information is not retained in the linked program and so uses no space at run time.

A second special function required of the linker is to be cognizant of the run time "allocate" function. The COL permits pointer variables to be initialized as part of the linking process, space for the object pointed to being allocated at that time. Details of this feature are presented in section 2.2.2.

In general, the linker is to be a high quality linker at the forefront of the present state of the art. As examples suggestive of what is expected of it, the linker is capable of pooling constants between modules, and it can relocate into any part of a word.

The linker can be controlled by the user using the "%linker" compiler directive, described in section 5.3.1.

#### 1.3.4: Run Time Environment

For the most part the COL assumes very little about its run time environment. In particular, it does not assume that there is an operating system in the computer. Of course, it takes advantage of such a system if it exists.



The language includes an "allocate" function and a "free" statement, to obtain space from a free-storage pool and to return to that pool such space when it is no longer needed. The run time environment must therefore include procedures to perform these functions. The "free" statement is described in section 3.4; the "allocate" function in section 4.3. The relevant run time procedures are described in section 6.5.

The programmer may specify to the compiler (using the "%check" compiler directive described in section 5.3.2) that certain semantic errors are to be detected by open code at run time. The run time environment must be prepared to deal with such errors once they have been detected.

#### 1.4: The "Tinman" and "Ironman" Proposals

In preparing the COL we have been aware of the present Department of Defense effort towards language standardization. In particular, we have given close attention to the so-called "Tinman" proposal [Fisher-76]. To the extent that the needs in communications are consistent with the goals of the Tinman, we have attempted to be consistent with it. However, we have felt free to differ where appropriate, in part because the Tinman has yet to become a standard within the DoD and in part because of our awareness of communication's special needs. However, we have not differed from the Tinman unless we felt that there was a real justification in doing so.

Recently the successor to the Tinman has appeared, the "Ironman" Report [DoD-77]. This was released too late to be able to affect the design of the COL. Appendix IV lists briefly the areas in which the COL design is at variance with Ironman criteria.

#### 1.5: About This Document

The intended audience for this document is the computer scientist with knowledge and skill in programming language design and at least a little experience in communications programming. This document is neither a tutorial introduction to the COL nor a reference manual for it. Although it does include most of the material that belongs in a reference manual, it contains more motivational and background material than is common in that sort of document. Our intention has been to make available to the language specialist not only the decisions we have made in designing the COL but also some of the thinking that led to those decisions.

Because this document is directed to the language design specialist, we have felt free where necessary to use the jargon of that discipline. We have included as Appendix III a glossary of the specialized terminology used in this report.

As a stylistic convention the bulk of this document is written in the present tense rather than the future, even though

it describes a language that has yet to be implemented. Thus there appear statements such as, "The compiler treats this construct by ...". The reader should interpret this as a statement of our intent about how any COL compiler is to operate. Clearly, no COL compiler yet exists.

#### 1.5.1: Organization of This Document

The next three chapters describe the bulk of the language, discussing respectively declarations, statements, and expressions. Chapter 5 discusses certain matters relating to lexical processing, and Chapter 6 covers various matters not discussed elsewhere. Finally, Chapter 7 presents some examples of COL code.

There are five appendices. Appendix I provides a quick reference to the syntax of the language, first listing all of the syntactic types and showing for each the page on which it is defined, and then listing the COL's complete syntax. Early readers of this document have found this appendix to be an invaluable reference aid. Appendix II contains a detailed example of a machine-level facility in COL for the SUE computer, along with a completely worked out example showing an implementation in the COL of parallel processes and co-routines. Appendix III defines that terminology used in this report that is perhaps not standard in the profession. Appendix IV contains an evaluation of the COL against the criteria of the "Ironman" proposal. Appendix V lists areas in which further language design seems called for.

## 1.5.2: Descriptive Method

The COL's syntax is presented using a slightly modified version of BNF. It is well known that the language designer has much freedom in the way the syntax equations are written, since there are many ways to specify a given language. In general, the emphasis here has been to ease the reader's task. In many cases a different choice would be more convenient for the compiler writer.

The usual angle brackets of BNF <like this> surround names of most syntactic units, but certain frequently used classes have names which are one or two upper-case letters. These are now listed, along with the section number in which each is described.

S	Ch3	statement
SD	Ch3	statement or declaration
E	Ch4	expression
NL	Ch4	constant expression, known at link time
NC	Ch4	constant expression, known at compile time
ID	5.1.1	identifier

A member of class S is any COL statement, while an SD is either a statement or a declaration. Syntactically E, NL and NC are the same, the only difference being that expressions NL and NC can be (and are) evaluated before run time. Expressions NC must be evaluable at compile time, while evaluation of expressions NL may be performed by the linker at link time. An ID is an identifier, the name of a variable or constant used by the programmer. The class <empty> is the empty set.

The word "class" as used above and throughout this document always means the fragment of language defined by a BNF definition.

Thus for example S and E and <conditional statement> each denote a class.

A few additional conventions are used. The notation

<foo> <fum> ...

stands for one or more <foo>s, separated by <fum>s, the sequence ending with a <foo>. For example,

S ; ...

stands for one or more statements separated by semicolons, so that any of

S  
S ; S  
S ; S ; S ; S ; S

are possible. Another example:

<digit> <empty> ...

defines (assuming that <digit> is a decimal digit) any string of decimal integers, such as "1" or "357" or "000111222". Specifically, "... " is a metalinguistic operator whose operands are the previous two items on the line. Note that both "." and ".." are used as operators in the COL (for structure reference and subrange declarations, respectively) and must be distinguished from "... " used as just described.

Syntax elements are always shown as separated by a single space. As explained in section 5.5.1 these spaces are not always required in COL coding. Each alternate of each syntactic class is shown on a separate line, so the usual "|" of BNF is not needed.



The following example should assist the reader in understanding the notation used. It defines the class <conditional statement>, and is an abbreviated version of the complete definition appearing in section 3.2.

```
1 <conditional statement> ::=
2   if E do S ; ... endif
3   test <alt list> endtest
4   test <altlist> otherwise S ; ... endtest

5 <alt list> ::=
6   E do S ; ...
7   E do S ; ... orif <altlist>

-- E ::=
   (see section Ch4)
```

Lines of syntax are numbered consecutively throughout this document. (The present example does not count.) An entry of "--" in the number field indicates a comment about a syntax class defined in a different section. A <conditional statement> as defined by the syntax above is either a one-armed conditional (line 2) or a multi-arm conditional (line 3). Line 2 generates the following <conditional statement>s, in which it is to be understood that each  $S_k$  is an instance of a statement  $S$  and each  $E_k$  of an expression  $E$ :

```
if E do S endif
if E do S1; S2 endif
if E do S1; S2; S3 endif
```

Any of the following is generated by line 3:

```
test E1 do S1 orif E2 do S2 endtest
test E1 do S1; S2 orif E2 do S3; S4 endtest
test E1 do S1 orif E2 do S2 orif E3 do S3; S4 endtest
```

The reader should be sure that these examples are understood.

### 1.5.3: Conventions Used in this Document

A few conventions are used in the examples of COL text presented in this document. It must be emphasized that they are not part of the language but are just used to assist the reader. First, it should be noted that the COL compiler does not distinguish between upper- and lower-case letters. Nonetheless, the examples in this document follow these rules:

- . All COL reserved words are spelled with lower-case letters.
- . All programmer-defined identifiers are spelled with upper-case letters.
- . All names of functions that provide access to specific hardware features are spelled with upper-case letters and a trailing underscore.

It is again emphasized that these are not part of the COL definition.

### 1.6: Character Set

The COL has been designed with the assumption that the full ASCII character set is available, but there is no construct in the language that is not readily expressible in the 64-character subset. See section 5.4 for further details.

## Chapter 2: Declarations

Two major decisions must be made early in the design process of any programming language.

Are declarations required? That is, must there be, for each variable to be used in a program, some sort of statement by the programmer that he is going to use that variable?

If declarations exist, is it enough just to state that the variable is to be used? Or must each such statement be accompanied by some information about the possible values the variable may have? That is, are type declarations required?

Most languages that require declarations also require that type information be a part of each declaration.. (BCPL and BLISS are notable exceptions, requiring that all variables be declared but not admitting into the language the concept of data type.) The COL requires both that every variable be declared and also that each such declaration specify the range of possible values the variable may have: the "data type" of the variable. A data type is a subset of all possible values that may be represented in the computer on which the program is to run, and specifying the type

of a variable in the COL is equivalent to stating that all values to be held by that variable will be from that set. For example, the COL declaration

```
declare ( X: 12 bit integer )
```

declares variable X to be an integer and informs the compiler that no more than 12 bits are needed to store the values to be held by X. (The relevant syntax is presented later in this chapter.) The COL compiler then insures that no non-integer value can be stored into X. At the user's option, it may also warn him about attempts to store into X any quantity that cannot be represented in 12 bits.

The COL is strongly typed: each identifier used by the programmer must be fully declared before it is used. Further, the compiler checks all such uses at compile time so as to insure that the type of the variable is consistent with its use. The intent in the design of the COL is that there be no subtle or non-obvious ways the programmer can circumvent the COL's type-checking rules. Although all of the restrictions can be circumvented by the programmer if he chooses, the mechanisms for doing so are designed to make what is being done immediately obvious to the reader of the program. (The "force" mechanism for getting around types is described in section 4.4.)

The strong typing means that EVERY access is checked, and checked completely. For example, the actual parameters in a

procedure invocation must agree in number and type with the formal parameters in the procedure declaration. This check is made even if the procedure declaration and the procedure invocation are not part of the same compilation(4). The term COERCION refers to a conversion from one type to another which is performed automatically by the compiler when the type checking rules do not match. The very few coercions that are done in the COL are described in section 6.1.5.

Each identifier used in a COL program denotes either a function or a routine or a scalar or an event or a macro, as suggested by the syntax shown below. This display format, used throughout this document to exhibit the COL's syntax, is described in section 1.5.2.

```
1 <declaration> ::=
2   <function declaration>
3   <routine declaration>
4   <scalar declaration>
5   <event declaration>

6   <macro declaration>
7   <data declaration>
8   <undeclaration>

-- <function declaration> ::=
   (see section 2.1)
```

---

(4) In this case the final check is made by the linker rather than the compiler. The important point is that the check is NOT made at run time. The program to be compiled supplies all of the needed type data for the compiler to make the checks; the linker merely checks that identical data were supplied to each module. (See section 1.3.3.)



```
-- <routine declaration> ::=
    (see section 2.1)

-- <scalar declaration> ::=
    (see section 2.2)

-- <event declaration> ::=
    (see section 2.3)

-- <macro declaration> ::=
    (see section 2.6)

-- <data declaration> ::=
    (see section 6.3.2)

9  <undeclaration> ::=
10  undeclare ( ID , ... )
```

The first four lines declare the kinds of identifiers mentioned above. The macro declaration provides for replacement of a name by a predefined string of text, with the possibility of supplying arguments when the macro is invoked. The data declaration provides information, usually machine or implementation dependent, about how data are to be laid out in the computer's memory; discussion is postponed to section 6.3, in which this and other machine dependent aspects of the COL are discussed.

Since the COL is block-structured, a variable once declared normally retains its association with that declaration throughout the remainder of the block(5) in which the declaration appears.

---

(5) The reader may for the moment think of a "block" as a sequence of declarations and statements separated by semicolons and surrounded by some bracket such as "begin...end", just as in ALGOL-60 or PL/I. There is a more precise definition in section 2.7.

In certain cases, though, it is convenient for the programmer to "undeclare" a variable, using the syntax of line 9. Doing so makes any use of that name in the rest of the block an error that is detected at compile time -- the name does not revert immediately to a value it might have had in an outer block but does so instead at the end of the block. This point is explained more fully in section 2.7.

Because the COL permits recursion, in both program and data structures, the order of processing of declarations is important. This matter is discussed in section 2.8.

## 2.1: Function and Routine Declarations

There are two kinds of subroutine in most programming languages, referred to collectively in this document as PROCEDURES. One kind of procedure is the FUNCTION, a procedure that returns a value and is used as part of an expression; and the other kind is a ROUTINE, a procedure that is executed for its effect and is used as a statement. In each kind of procedure, parameters may be passed as part of the invocation. Following ALGOL-60, the term FORMAL PARAMETER is used for the identifier that appears in the procedure declaration as the dummy argument, and the term ACTUAL PARAMETER for the argument actually supplied when the procedure is invoked.

Note the use of comments in the syntax below. Comments are always lined up vertically about two-thirds of the way across the page and start with "//". Sometimes the reader of this document must use a bit of care to distinguish comments from long syntax lines.

```

11 <function declaration> ::=
12   <mode> function ID <fpl> : <storage> <type> ; SD ; ...
                                   endfunction
13   forward function ID <fpl> : <storage> <type>

14 <routine declaration> ::=
15   <mode> routine ID <fpl> ; SD ; ... endroutine
16   forward routine ID <fpl>

17 <mode> ::=                                // compilation mode
18   open                                     // compile open as a macro
19   closed                                  // closed subroutine
20   <empty>                                 // default is either

21 <fpl> ::=                                // formal parameter list
22   ( <fp> , ... )                          // formal parameters
23   ( )                                     // no parameters

24 <fp> ::=                                // formal parameter
25   <call type> ID , ... : <storage> <type>

26 <call type> ::=                          // how param is passed
27   value                                  // by value
28   ref                                   // by reference
29   variadic <call type>                  // variadic procedure
30   <empty>                               // default

-- <storage> ::=
   (see section 2.2.1)

-- <type> ::=
   (see section 2.4)

-- SD ::=
   (see section Ch3)

```

All formal parameters must be fully specified, with all type information given explicitly.

Note that there are two types of declaration for each of function and routine, the first of which has a body (a sequence of statements and declarations), while the other starts with "forward" and has no body. The declarations with a body do what is expected in a programming language: The identifier that is the name of the procedure is declared as a constant identifier, associated within its scope with the formal parameter list and body in the usual way. For example, the declaration

```
function Successor(N: integer): integer
    resultis N+1
endfunction
```

defines a function named Successor which returns the integer successor of its integer argument.

The word "forward", as used in lines 13 and 16, provides for the case in which it is necessary to invoke a procedure within a module before it is declared. This is necessary only for mutually recursive procedures, the case in which each of two procedures invokes the other one. An example is presented in section 2.7.

Note that the body of a function is a sequence of SDs separated by semicolons. (Recall from section 1.5.2 that an SD is either a statement or a declaration.) An instance of the statement form "resultis E" appearing in the body of a function causes evaluation of the function to cease immediately, with E

taken as the value of the function. There may well be multiple occurrences of "resultis" in the body. It is an error detected at compile time if control "falls off the end" of the function body.

The body of a routine is a sequence of SDs separated by semicolons. The statement "return" appearing in a routine body results in a return from the procedure to the calling point. "Falling off the end" of the routine body also causes such a normal return.

Note that a function or routine declaration may optionally be preceded by a <mode>, which may be "open" or "closed". The mode "open" means that the body is compiled as open code at each place of invocation (somewhat like a macro but with certain important differences to be explained), while the mode "closed" means that a closed subroutine with a calling sequence is compiled in the usual way. The default if no mode is given is that the compiler chooses. The programmer can often improve the time efficiency of his program, usually at the cost of space efficiency, by using an open procedure. However, there are situations in which open coding improves both space and time efficiency. Consider for example a procedure in which a branch is made based on the value of a parameter. If the value of the actual parameter is known at compile time and the procedure is compiled open, the compiler will know which way the branch must go and will not compile the branch(es) that cannot be executed.



Although changing the mode of a procedure may well impact on the time/space efficiency of the compiled code, it is the intent of the COL design that such a change is to have no other effect on the semantics of a procedure. This decision has an impact on the binding to declarations of variables that occur free in the procedure body(6). Consider a variable that is used in an open procedure but not declared in it. Since the semantics of a procedure is to remain unchanged regardless of the mode, such an identifier is bound to the declaration in effect at the point where the open procedure is declared, and NOT at the point where it is invoked. If binding at the point of invocation is required, macro substitution as described in section 6.1.2.3 should be considered. See also the further discussion of open procedures in section 6.1.2.2.

Empty parentheses are used in line 23 as part of the declaration of a function or routine that takes no arguments. As is seen in sections 4.4 and 3.1, empty parentheses are also used in invoking a function or routine which takes no arguments.

The same ID may not be used in more than one parameter position. An exception to this rule is that the same ID may be used in more than one flexible array limit, with the meaning that the actual parameters supplied must have the same corresponding

---

(6) Terms such as "free" are defined in Appendix III.

limit. See section 6.1.1.2.

Note in line 25 that a <storage> may be specified for a formal parameter. Normally parameters are passed on the stack, but for efficiency or other reasons the programmer may specify some other place. The syntax for <storage> appears in section 2.2.1, which should be looked at. As used in the context of a formal parameter, the possibilities are as follows:

- . static. A single location is assigned by the compiler to hold the parameter, that location being used for no other purpose. This is probably a bad choice for a re-entrant(7) procedure, since the parameter is shared at all levels of re-entrancy.
- . dynamic. This is the default if no <storage> is specified, and the parameter goes on the stack as previously described. There is a separate instance for each level of re-entrancy.
- . location. The effect is the same as for static, except that the programmer rather than the compiler selects the location to be used. It is of course implementation dependent.
- . register. The parameter goes into a specified machine

---

(7) This term is defined in Appendix III.

register. This is of course implementation dependent.

Note that all but dynamic (and <empty>) put the parameter into static storage, with implications on the scope rule as described in section 2.7.

Note on lines 12 and 13 that a <storage> may precede the <type> that describes the value returned by the function. This <storage> permits the user to specify where the returned value is to be stored. The possible values are as follows:

- . <empty>. The default if no <storage> is specified is determined by the compiler. In general, the programmer need not care.
- . register. The value is returned in the designated live register. This is of course implementation dependent.

The other possibilities of <storage> are not permitted in this context. If the returned value of a function is a flexible array, the only permitted storage class is dynamic (or <empty>).

The <call type> specifies how the actual parameter is to be made available to the procedure. The default if no call type is specified is read-only. The actual parameter may or may not be copied, the choice being made by the compiler and not of concern to the user. However, it is an error detected by the compiler for the programmer to attempt to update the formal parameter. (That

is, the formal parameter may not appear on the left side of an assignment statement, nor may it be an actual parameter to another procedure unless the call type is by value or read-only.) This call type is most efficient and should usually be used unless one of the other two is required by the logic of the program. Call by value means that the procedure is supplied a copy of the actual argument, so that updating the formal parameter is permitted but has no effect on the actual parameter. Call by reference means that the procedure is supplied the address of the cell containing the actual parameter, so an update of the formal parameter causes the contents of the actual parameter to be changed.

The call by value corresponds exactly to the same concept in ALGOL-60, while call by reference corresponds to the parameter passing mechanism used in FORTRAN and other languages. There is nothing in the COL corresponding to ALGOL-60's call by name. If the formal parameter is marked as call by reference, it is an error detected at compile time if the actual parameter is not addressable. A constant or a structure component not aligned on address boundaries or a calculated expression such as  $X+1$  is such a non-permitted actual. (The term "addressable" is defined in Appendix III.)

If the actual parameter is an aggregate and the call is by value, the entire aggregate is copied. However, if a component of an aggregate is a pointer, the pointer itself is copied but the

object it points to is not. Inasmuch as calling a large aggregate by value can lead to an expensive copying operation, the programmer should refrain from calling aggregates by value unless his application really requires doing so.

A parameter of type general may not be called by value, as the compiler cannot know how much space it requires.

The syntax of line 29 is used to declare a variadic function or routine, one that takes a varying number of arguments. The declaration and use of variadic procedures are described in section 6.1.1.3.

## 2.2: Scalar Declarations

A scalar is a name that represents either a variable or a constant or a type.

```
31 <scalar declaration> ::=           // declare scalars
32     declare ( <decl> ; ... )

33 <decl> ::=                           // declare a single scalar
34     <variable decl>                 // it's a user variable
35     <constant decl>                 // it's a constant
36     <type decl>                     // it's a name for a type

-- <variable decl> ::=
    (see section 2.2.1)

-- <constant decl> ::=
    (see section 2.2.2)

-- <type decl> ::=
    (see section 2.2.3)
```

An identifier ID is a name that the programmer deals with in his



programming. A VARIABLE denotes a value which can be changed as the program executes. A CONSTANT is an identifier whose value the compiler knows; any appearance of that identifier is treated by the compiler as if the value itself had appeared. A <type decl> declares an identifier which denotes a type. These concepts are now treated in turn.

### 2.2.1: Declaration of Variables

To declare a variable, the programmer must specify its type. In addition he may optionally specify any or all of storage class, size, and initial value.

```

37 <variable decl> ::=                // declare a variable
38     ID , ... : <volatility> <storage> <type> <init value>

39 <volatility> ::=                    // accessed each time?
40     volatile
41     <empty>

42 <storage> ::=                       // storage class
43     static                         // static storage
44     dynamic                        // storage on the stack
45     location ( NL )                // resides at address NL
46     register ( <register> )        // resides in register
47     <empty>                       // default is dynamic

-- <type> ::=
    (see section 2.4)

48 <init value> ::=
49     initially E
50     <empty>                       // no value specified

-- <register> ::=
    (machine dependent -- see section Ap2.1.1)

```

Each variable has associated with it a set of attributes,

specifying its volatility (described in the next paragraph), where the datum is to be stored (its storage class), the data type to be associated with the variable, and the initial value (if any) to be assigned to the variable whenever its storage is assigned. These attributes are specified as shown above by <volatility>, <storage>, <type>, and <init value>, respectively.

If an item is volatile, it must be accessed from memory each time it is mentioned. For example, a variable corresponding to a machine register used to control I/O must be accessed each time it is mentioned, since it may well have a different value each time. In the absence of this attribute the code optimizer may produce incorrect code. This point is addressed further in section 4.5, which considers the implication of the code optimizer on coding style.

The definition of <register> is of necessity machine dependent and is not specified as part of the COL's definition. An example of a possible syntax for <register> is given in section Ap2.1.1.

#### 2.2.1.1: Storage Class

The storage class determines where an item resides during execution. <storage> is used in three places in the syntax in addition to its usage in the preceding section. In section 2.1 it is used on line 25 to permit the programmer to specify where a parameter is to be passed to a procedure and on lines 12 and 13 to

specify where the value returned by a function is to be stored, and in section 6.3.2 it is used in the data declaration to specify just where static data are to be stored.

There are five possibilities for <storage>. The two common ones are on the stack (dynamic storage) and static. The former provides storage that comes into existence after the block in which the declaration appears is entered and that goes out of existence when the block is exited. (This is equivalent to the PL/I automatic attribute, or to any ALGOL-60 variable not declared "own".) A static variable is assigned space before the start of program execution and always resides in that space. It therefore retains its value over exits from the block in which it is declared. (This the ALGOL-60 "own" concept and is the usual FORTRAN convention.) Consider a variable declared within a re-entrant procedure. If it is static, then there is only a single memory location provided for its storage, that location being shared by all invocations of the procedure. If it is dynamic, each invocation has its own instance of the variable. Both possibilities are useful; the programmer must be sure he understands the issues when he makes his choice.

As mentioned towards the end of section 1.1.2, some implementations of the COL may not have a stack. In such implementations there is still a distinction between static and dynamic storage, since for the latter the compiler may elect to

share storage for parallel blocks, whenever practical.

The attribute "location (NL)" specifies that the variable is to reside in machine location NL. (Recall that an expression NL is a constant expression whose value need not be known until link time.) This is of course a machine-dependent concept. It is useful, for example, for architectures in which I/O or interrupt handling are performed by reading or writing specific memory locations.

The register attribute (line 46) specifies that the variable is to reside in a machine register. The details are of course implementation dependent. A possible syntax for <register> is discussed in section 6.3.2, and a proposed syntax for the Lockheed SUE is in Ap2.1.1. In many implementations it is not appropriate to specify a <size> for a register.

In connection with the two previous paragraphs, the COL regards location and register declarations as placing the items so described into static storage. This point is important with respect to accessing such items within the bodies of procedures declared within their scope. See section 2.7 for further discussion.

#### 2.2.1.2: Type

The concept of data type partitions the class of values that can be manipulated by the running program into useful disjoint

sets, such as integers, logical data, booleans, etc. Details of the types available in the COL and the syntax used to describe them are provided below in section 2.4.

#### 2.2.1.3: Initial Value

Specifying an initial value for a variable means that the variable is given that value each time storage for it is assigned. If the variable is dynamic (resides on the stack), then it is initialized each time the declaration is executed and the value is calculated at that time. If the variable is static, it is initialized only once and the initial value must be evaluable before execution of the program, either at compile time or at link time. The exact details of initialization of static storage are implementation dependent. If the variable has storage class location, initialization is the same as for any other static value. For storage class register initialization occurs on each execution of the declaration, as for dynamic storage.

If no initial value is specified, the programmer can make no assumption about the variable's initial value. Further, a dynamic variable does not in general retain its value between exit and reentry of the block in which it is declared.

#### 2.2.2: Declaration of Constant Names

In many programming applications it is convenient to specify that a given identifier has a constant value known to the compiler. Further, each ID to be used as a label constant must be



declared before use so as to indicate its scope. (A label constant is an identifier followed by a colon that labels a statement.)

```
51 <constant decl> ::=
52     ID , ... = NL           // constant value
53     ID , ... = label       // label constants
```

A declaration of the form on line 52 defines one or more identifiers to have a constant value. Thus

```
declare ( M = 4 )
```

specifies that M is an integer constant with value 4, and each appearance of M is treated as if 4 had been written. Note that an arbitrary expression can be used as the value, just so long as it can be computed at compile time. (The syntax permits this evaluation to be deferred till link time; this point is discussed in the next paragraph.) A constant declaration is processed by evaluating the constant expression, and then associating that value with the name. Note that the expression is NOT reevaluated each time the name is used, so binding of free variables that appear in it is as of the place of the declaration. It is a requirement of any COL compiler that evaluation at compile time is to yield exactly the same value as the corresponding evaluation performed at run time, at least insofar as this is possible. If the compiler is a cross-compiler (running on one machine compiling code for another), integer arithmetic must be simulated exactly. In some cases it may not be feasible to simulate exactly floating

point arithmetic, since the documentation of many CPUs is so poor that one cannot deduce the fine details of floating point arithmetic algorithms. The compiler must do the best job possible.

Each ID declared as a constant has a type, just as does each ID which is a variable, and the COL's type-checking rules are enforced on that ID. The type of a declared constant is the type of the expression. If a different type is wanted, the "convert" form may be used. In particular, "convert" must be used to declare a constant of a "different" type, as that term is defined in section 2.2.3.

Postponement of evaluation of the constant declaration until link time is permitted in only one way in the COL. The variable must be a pointer, and the expression NL must invoke the built-in "allocate" function, described in section 4.3. The effect is that the identifier is declared to be a constant pointer, always pointing to the same location, and the "allocate" is processed as part of the linking process to determine just where the item is to reside. For example, consider the following declaration:

```
declare (P = allocate(integer: 0))
```

Here P is a pointer to an integer, and when the program begins P is permanently fixed to point to a location which is initialized to zero. The value P points to may be changed, but P itself may not be. Another way to think of this is that an assignment

statement of the form

P := ...

is detected as an error at compile time while

P@ := ...

is correct.

The syntax on line 53 provides for declaring identifiers to be used to label statements. Each such identifier must be declared before it appears either as a label on a statement (followed by a colon) or as the identifier after "goto". Such a declaration alerts both the compiler and the reader to the impending use of a label. Although we are not so rabid on the issue of structured programming as to eliminate labels entirely from the COL, we have added this syntax at least in part to help make programs with labels as easy as possible to read. (It bothers us very little that we have thereby made them slightly harder to write.)

### 2.2.3: Type Declaration

If a (perhaps complicated) type is to be used repeatedly in a program, it is frequently convenient to give that type a name by which to refer to it.

```
54 <type decl> ::=                // declare name for a type
55     ID , ... is <type>          // synonym for a type
56     ID , ... is different <type> // same properties

-- <type> ::=
    (see section 2.4)
```

An identifier so declared is treated thereafter as if the entire <type> had been written out; it is sort of an abbreviation. However, any free identifier that appears in it, as for example an array limit or a size, is evaluated at the place of the declaration of the type. (This point is elaborated further in section 6.1.2.1.) If the programmer prefers binding at the invocation place, the macro facility of section 6.1.2.3 should be considered instead.

The form of definition on line 55 declares a synonym for a type, the new type being treated by the compiler exactly as if the old type had been written out fully. The form on line 56, on the other hand, creates a new type with all of the same properties as the old type but not interconvertible. Consider

```
declare
(  T1 is integer;
  T2 is different integer
)
```

Variables of type T1 are treated in all ways as if they had been declared to be of type integer, while the value of a variable of type T2 cannot be used in a context in which an integer is expected. However, an integer constant (such as 3 or 27) may be used in a context in which type T2 is expected, and all the operators that operate on integers operate equally well on type T2. If several IDs are listed, as provided for in line 56, they are all synonyms for one another although different from the type on the right. Thus the effect of

A, B is different integer  
is different from that of

A is different integer  
B is different integer

since in the former A and B are synonymous types and they are not  
in the latter. This point is addressed further in section 6.1.5.  
However, the example in Figure 2-1 maybe helpful. Here each of

-----

```
declare
( APPLES is different integer;
  ORANGES is different integer;
  P: APPLES;
  R: ORANGES
)

...

... P + R ... // APPLES + ORANGES
```

Figure 2-1: Adding Apples and Oranges

-----

APPLES and ORANGES is a type different from integer, and variables  
P and R are of type APPLES and ORANGES, respectively. The  
compiler detects as an error the attempt to add APPLES to ORANGES.



### 2.3: Event Declarations

An event declaration provides a mechanism permitting the programmer to specify that the occurrence of some (usually machine-dependent) event is to be checked for, with provision for branching later to determine whether or not that event has taken place.

```
57 <event declaration> ::=
58     check <event>

59 <event> ::=
60     ID
```

The specific IDs that are permitted <event>s are specified as part of each implementation. However, some are defined in all COL implementations, an example being the "conversion" event that the programmer may use to detect errors in the conversion functions described in section 4.4.

The effect of such a declaration is twofold: It informs the compiler that within the scope of the event declaration the particular event is to be checked for, and it declares a dynamic boolean variable initialized to false with the same name which the programmer may evaluate to determine whether or not the event has taken place. For example, the declaration

check overflow

might well direct the compiler to check for overflow in arithmetic operations within its scope. The boolean variable overflow is thereby declared, and the subsequent statement

if overflow do ...

may be used to determine whether or not an overflow has occurred. Specifically, this test determines whether or not an overflow has occurred in the part of the text that appears between the event declaration and the test. An overflow in a procedure invoked in this area cannot be detected this way, since the scope of the variable "overflow" is limited in the usual way. See section 6.1.4 for further discussion.

## 2.4: Types

The COL provides the programmer a rich selection of data types to use in constructing his programs. This section describes the various types and gives the syntax used by the programmer to specify them.

```
61 <type> ::=
62     <size> <unsized type>

-- <size> ::=
    (see section 2.4.1)

63 <unsized type> ::=
64     <basic type>
65     <aggregate type>
66     <other type>

-- <basic type> ::=
    (see section 2.4.2)

-- <aggregate type> ::=
    (see section 2.4.3)

-- <other type> ::=
    (see section 2.4.4)
```

Issues relating to `<size>` are presented in the following subsection, and the three `<unsized type>`s are discussed in the subsections after that.

#### 2.4.1: Size

Any data type may have associated with it a `<size>`, which specifies the amount of storage required for a datum of that type.

```

67  <size> ::=                                // size of a datum
68      <integer> word                        // size in words
69      <integer> byte                        // <integer> bytes
70      <integer> bit                         // <integer> bits
71      word                                 // 1 word

72      byte                                // 1 byte
73      bit                                 // 1 bit
74      <empty>                             // implementation dep

--  <integer> ::=
    (see section 5.1.2)

```

The type of a variable does not in and of itself specify the minimum size of the associated datum in storage; instead, this attribute may be specified by the programmer using the `<size>` attribute. For example,

```
1 byte integer
```

specifies that the integer is to be stored in (a minimum of) one byte. The compiler may elect to allocate more space if that is advantageous, unless the programmer has specified otherwise (as in a packed aggregate). It is a detected error at compile time if the size is not an appropriate one for the object computer. For example, the COL does not simulate multi-precision arithmetic if

it is not available in the hardware. Details of the interactions of various sized operands on the COL's operators, including assignment and parameter passing, are in section 4.2.2.

The size can be specified in terms of words, bytes or bits. The number of bits in a word or in a byte is of course dependent on the characteristics of the object machine. If no <size> is specified, then a machine dependent default is used. This default size provides enough bits to represent the type involved in the object machine.

The <integer> may be omitted if 1 is intended, so that the two declarations

```
word integer
1 word integer
```

are equivalent.

Note that the syntax requires that the size be an <integer> rather than permitting any expression NC that can be evaluated at compile time. This seemingly arbitrary restriction is an unusual one in the COL and deserves an explanation. Replacing <integer> by NC causes serious parsing problems, making it extremely difficult to produce an LALR(1) grammar(8) for the resulting language. The programmer may use the macro facility to

---

(8) An LALR grammar is one which can be parsed by a particular type of efficient parsing algorithm. The COL has been designed with this type of grammar in mind. See [Morgan-77].

parameterize a data declaration.

#### 2.4.2: Basic Types

The basic types are the building blocks from which all the more complex types are built.

```
75 <basic type> ::=
76     integer           // integer
77     float             // floating point quantity
78     float ( NC )      // float, with precision
79     logical           // bit data

80     <charset> char     // character data
81     boolean           // true or false
82     interlock         // locked or unlocked
83     condition         // for Zahn signal

84 <charset> ::=
85     ID                // special set
86     <empty>           // default set, ASCII
```

These types have essentially the conventional meanings usually ascribed to them in most programming languages. However, a few words about each are appropriate.

An integer is a signed quantity with default size appropriate to the implementation.

A floating point quantity is represented in some machine dependent way with a mantissa and an exponent. The optional parenthesized quantity specifies the minimum required precision in decimal digits. For example, the type "float(3)" means that a space at least 10 bits wide is required to store the mantissa of such an object. It is an error detected at compile time if the



<size> specified is inadequate.

A logical datum is a string of bits to be manipulated as a unit.

A datum of type character is from a subrange. (The type subrange is described in section 2.4.4; briefly, it is some subset, usually of the integers, specified by the programmer.) The character encoding is defined by the character set in use. The default character set is ASCII, although the COL provides a mechanism for the user to specify alternate character sets of his choosing. (This is the "% chars" compiler directive described in section 5.3.3.) Briefly, a compiler directive is provided to permit a user to declare a new character set and to specify its collating sequence. Such a character set is effectively a new type.

A boolean variable can take on only the values true and false and can be stored in a single bit. The amount of storage actually used is an implementation decision.

An interlock is used for synchronizing parallel processes; details are provided in section 3.4.3.

The <type> "condition" corresponds to a Zahn signal and may be used only to describe a formal parameter. See section 3.4.2.

Details about the semantics of integers and of fixed and floating point quantities as used in expressions appear in section 4.2.2, in which the semantics of the COL's operators is presented.

#### 2.4.3: Aggregates

An aggregate is a collection of items built up out of simpler components. An array is an aggregate each of whose components is of the same type; a structure is an aggregate whose components may be of differing types. The elements of an array are selected by an integer subscript whose value need not be known until run time; the elements of a structure are selected by name, the selection being known at compile time.

```

87 <aggregate type> ::=
88     array <array bound> , ... of <type>
89     structure <s-mode> <field list>
90     packed <aggregate type>           // minimize space
91     unpacked <aggregate type>         // minimize access time
92     parallel <aggregate type>

93 <array bound> ::=                      // subscript limits
94     <discrete type>
95     ID

-- <discrete type> ::=
   (see section 2.4.4)

96 <field list> ::=
97     ( <field> ; ... )

98 <field> ::=
99     <s-mode> ID : <volatility> <type> <init value>
100    : <type>                               // unnamed field
101    selection ID into ( <var-field> ; ... )
102    <s-mode> <declaration>                 // encapsulated data type

103    <assertion>                           // invariant of capsule
104    start SD ; ... endstart                // initialization

```

```
105      finish SD ; ... endfinish          // finalization
106  <var-field> ::=                          // variant field
107      <case label> <field list>
108  <case label> ::=
109      <case> : ... :
---  <case> ::=
      (see section 3.4.1)
110  <s-mode> ::=
111      public
112      private
113      <empty>
---  <volatility> ::=
      (see section 2.2.1)
---  <type> ::=
      (see section 2.4)
---  <init value> ::=
      (see section 2.2.1)
---  <assertion> ::=
      (see section 6.4)
```

This syntax provides for the declaration of both arrays and structures, most of the syntax being for the latter. The array syntax is on lines 88 and 93 to 95. Note that multiple <array limit>s are permitted in an array declaration, as shown on line 88. This syntax provides an abbreviation for an array, each of whose elements is itself an array. For example, the type

array[1..10] of array[-3..3] of integer

is indistinguishable within the compiler from the type

array[1..10], [-3..3] of integer

but the latter is a syntactic sugaring for the former. In section 4.3 it is pointed out that a similar syntactic sugaring is

available to access an element of such an array.

Note that a structure is made up of `<field>s` separated by semicolons. In many cases the structure contains only data, using the syntax on lines 99 to 101. A structure that uses the syntax on lines 102 to 105 is said to define an ENCAPSULATED DATA TYPE, a mechanism that provides a way to specify a data type in such a way that it is not possible for an unauthorized procedure to alter its components. This ability is extremely useful if verification of the resulting program is a requirement. The feature is described in section 6.1.3, the remaining description in this section being for the simpler case. Since in such cases the only useful `<s-mode>` is `<empty>`, discussion of `<s-mode>` also is postponed to section 6.1.3.

A structure item (line 99) specifies a named field with a given type and size, as well as an optional initial value. The interpretation is as follows: Each field in the structure is assigned consecutive space in a way defined by the packed and unpacked and parallel options (described below). The size of a "selecton" is the maximum size of one of its entries. The fields following a "selecton" have an offset which is determined by the maximum size of the "selecton" entry. Hence every offset and position can be determined at compile time. The ID in the "selecton" must be one of the identifiers defined as a `<field>` at or above this level of "selecton" recursion, and the values used

as case constants must be possible values of that field.

If a field has an <init value>, that value is calculated at the time storage for the field is allocated. Thus if the structure is part of a type declaration, then the initial value is calculated each time a variable of that type is declared. The expression used in the initial value may not involve any dynamic variables. Further, all binding is as of the place of declaration of the structure.

An example of a structure using "selecton" appears in Figure 2-2. This(9) defines a structure STUFF which includes a TYPE field which is either PIP or POP or PUP, an EXTENT field which is an integer, and a LINK field which is a pointer to another STUFF. In addition, if the TYPE field has the value PIP there are to be fields I1 and I2 of type integer, and if it has value PUP there is to be U1 of type integer and U2 of type boolean. The intent is that fields I1 and I2 should be accessed only in code that has determined that the TYPE field has value PIP, and fields U1 and U2 from code in which it has value PUP. This intent is suggested by the two conditional statements shown at the bottom of the figure.

If the programmer wants to specify the details of the packing, he may use unnamed fields (line 100) to leave space in

---

(9) The example uses the COL comment convention that the mark "//" and all text to its right on a line are ignored. The COL's comment conventions are described in section 5.5.2.



```

declare
(  TYPES is (PIP, POP, PUP);    // A new enumeration type.

  STUFF is structure
  (  TYPE:  TYPES;
    EXTENT: integer;
    LINK:  pointer STUFF;
    selecton TYPE into
      (  case PIP: (I1: integer; I2: integer);
        case PUP: (U1: integer; U2: boolean)
      )
    )
  )

  V1, V2: STUFF    // 2 variables of type STUFF
)

...
if V1.TYPE = PIP do ... V1.I1 ... endif
if V1.TYPE = PUP do ... V1.U1 ... endif

```

Figure 2-2: Example of a Structure Declaration

-----

the structure.

The compiler must decide how to store each aggregate. If not otherwise instructed it will choose a reasonable compromise that does fairly well on both storage space required and access time to the components. If the programmer specifies that the aggregate is to be packed (line 90), then the compiler will pack the data into minimum space with little regard to access time. Contrariwise, specifying unpacked (line 91) causes the compiler to minimize access time without much regard for the space cost. In neither case does the compiler completely ignore the undesired property --

it refrains from choosing a completely disastrous allocation. The programmer may direct the compiler to go all of the way by using "packed" or "unpacked" twice. In all cases, however, the elements of the structure are stored in the order specified in the structure declaration. }

An example may help:

```
declare ( F00: array[1..30] of 5 bit logical )
```

This declares F00 to be an array of 30 items, each of which is a 5-bit quantity. The compiler selects a representation which is reasonably efficient with respect to both space and time. For example, if the object machine has a 16-bit word and has opcodes to access half-words, each item might be put in an 8-bit half-word. Specifying "packed" before "array" directs the compiler to do a better job of optimizing space. In this case it might store three items per word, wasting one bit per word. Using "packed packed" causes dense packing with some items overlapping word boundaries. Using "unpacked" would (if there were a time advantage) cause the compiler to store each item in a separate word. There seems to be nothing to gain in this example from "unpacked unpacked".

It is important to realize that "packed packed" guarantees that the structure is stored exactly as it is declared, regardless of how hard it is to access its fields. Consider a structure declaration that describes the fields of a message transmitted

between computers. If the "packed packed" attribute is used, the same declaration may be used to describe the message on both computers, even if they have differing word lengths. An interesting example is the host to host message format used in the ARPANET, a declaration for which is shown in Figure 2-3. This is

```

-----

declare
( HOST_HOST_MSG is packed packed structure
  ( LEADER: HOST_HOST_LEADER
    M1:      8 bit logical  // padding, always zero
    S:       8 bit integer  // connection byte size
    C:       16 bit integer // byte count
    M2:      8 bit logical  // padding, always zero
  )

  HOST_HOST_LEADER is packed packed structure
  ( PRIORITY: 1 bit boolean
    IMP_X:    1 bit boolean
    TRACE:    1 bit boolean
    OCT:      1 bit boolean
    TYPE:     4 bit [0..15]
    ORIG_HOST: 2 bit [0..3]
    ORIG_IMP:  6 bit [0..63]
    ID:       12 bit logical
    SUBTYPE:  4 bit [0..15]
  )
)

```

Figure 2-3: Host to Host Message Format in ARPANET

the declaration of two types, HOST\_HOST\_MSG for the header of a message, and HOST\_HOST\_LEADER for the first 32 bits of that header. (Note that the first field of HOST\_HOST\_MSG is type HOST\_HOST\_LEADER.) If information is stored into the fields of such a header in one computer using this declaration and the message then sent over the net to another computer, the same values may be retrieved using the same declaration, even if the computers have different word lengths. Note that both structures are "packed packed", and note that the boolean fields are specified as "1 bit" so as to give the compiler no freedom at all to optimize anything. Note also that the originating IMP field ORIG\_IMP is 6 bits wide and has the type of a subrange from 0 to 63. Had it been "6 bit integer" the compiler would interpret the left bit as a sign, giving a range from -32 to 31. (The declaration "6 bit [-2..40]" is an error.) This kind of declaration should be used only in situations, such as this one, in which it is necessary for the application, because it can lead to quite inefficient code. For example, on an a machine with an 18-bit word the ID field of the leader goes over word boundaries.

The attribute "parallel" (line 92) is of use in dealing with an array of structures. Consider the following example:

```
declare
  ( FOO: array[1..30] of
      structure(A: integer; B: integer)
  )
```

This is an array of structures, each element of the array

occupying two words. In a word-addressed machine the array subscript must be doubled as part of each array access. Preceding the word "array" by "parallel" changes the storage for F00 but does not change the notation for accessing its elements. F00 is then implemented as two parallel arrays. The important part of the parallel attribute is that using it does not change the code written to access the elements of the aggregate, so the decision about how to store an array of structures can be postponed until quite late in the design process.

#### 2.4.4: Other Types

A few types remain, types that do not fall into any convenient class.

```
114 <other type> ::=
115     pointer <type>                // pointer
116     function <fpl> : <storage> <type>
117     routine <fpl>
118     general                        // union of all types

119     <discrete type>
120     ID                            // previously declared

--- <fpl> ::=
    (see section 2.1)

121 <discrete type> ::=
122     [ <limit> .. <limit> ]        // subrange
123     ( ID , ... )                // enumeration of scalars

124 <limit> ::=
125     E                            // subrange limit
126     ? ID                        // constant limit
                                // run time limit
```



A pointer is very much like an address, but to meet the COL's strong typing requirement the type of the object pointed to must be included as part of the type of each pointer. The COL provides the postfix operator "@" to access the object pointed to by a pointer. (See section 4.4.) Further, the equality and relational operators are defined on pointers. Thus pointers are ordered, although the ordering is implementation dependent. Only pointers of the same type may be compared, it being a compile error to attempt to compare pointers of differing types. The justification for defining pointers to be ordered, even though the order itself is not defined, is presented in section 4.2.2.4.

The syntax on lines 116 and 117 provides for variables of type function and routine, respectively. Consistent with the COL's strong typing, the types of all of the arguments (as well as the returned value of a function) must be included in the type of the variable. For example,

```
declare ( F: routine(M: integer, N: integer) )
```

declares F to be a variable whose value can only be a routine which takes two integer parameters. The compiler reports as an error an attempt to assign to F any value of differing type.

Note that there is no type "label", so that the COL does not permit a variable whose value is a label nor a procedure parameter which is a label. The syntax on line 53 is used to declare that IDs are label constants.

The type "general" (line 118) is the union of all possible types. There are only two contexts in the COL in which an item whose type is "general" may be used. Such an item may be declared only by being a formal parameter of a procedure, and it may be used only as an argument to force. (The latter is described in section 4.4.) Any use of type general is machine dependent and is flagged as such in the compilation listing, since the program is evidently using some property of the machine representation of an object.

There are two kinds of discrete type: the subrange (line 122) and the enumeration (line 123). A subrange is a contiguous finite subset of some set, specified by exhibiting the subset's limits. The sets that may be subsetted from to form a subrange are the following:

- . the integers. Examples are [1..10] or [-7..7].
- . characters. In ASCII the set [a..z] represents the lower-case letters, and [0..9] the digits.
- . an enumeration.
- . a subrange. Any subrange may be further restricted.

Thus the declaration

```
declare (Q: [1..10])
```

indicates that Q is to take on integer values between one and ten,

inclusive. Q may be used in all contexts in which an integer may be used. In particular, Q may appear on the left side of an assignment statement in which an integer expression appears on the right. The compiler may elect to use only enough storage for Q (four bits in this case) to store the specified values, but it does not in general make further checks that the value to be stored into Q is in range. (The user may direct that such checks be compiled, using the "% check subrange" compiler directive as specified in section 5.3.2.)

An enumeration is a listing of scalars. For example, the declaration

```
declare (R is (A, B, C, D) )
```

declares R to be a new type, such that a variable of that type can take on only the four values listed. An additional effect of this declaration is to declare each of A, B, C and D as constants, with values that are not available to the programmer. That is, the four identifiers can be used only in connection with the associated enumeration type. The elements of an enumeration are ordered and can therefore be subsetted to form a subrange.

Note on line 125 that any expression E may be used as the limit of an array. Except in certain very special contexts, the limit must be known at compile time (as if the syntax used "NC" instead of "E"). The exception is for an array stored in free space; see section 6.1.1.2 for details.

The question mark notation of line 126 defines a flexible array. Such an array may be used only in a formal parameter list or as the value returned by a function. See section 6.1.1.3 for further details.

## 2.5: Examples of Declarations

Line 12 provides the syntax for function declarations. Here is an example:

```
function Square(n: integer): integer;  
    result is n*n  
endfunction
```

This defines "Square" to be a function which returns an integer value which is the square of its argument. Another example:

```
function FOO(value z: integer): boolean;  
    statements & declarations...  
    result is true;  
    ...  
    result is (complicated boolean expression...)  
endfunction
```

The following example illustrates some of the possibilities of type declarations.

```
declare  
( A, B, C: static integer;    // 3 static integers  
  N : integer initially 7;    // initial value  
  P = 29;    // compile-time constant  
  L1, L2: interlock;  
  FOO is [1..100]    // declare a subrange  
)
```

The first four lines(10) declare scalar variables, the fourth

---

(10) The mark "///" in COL text introduces a comment which extends to the end of the line. (See section 5.5.2.) Note also that semicolons appear after statements in the preceding example and

declaring two variables to be used as interlocks. (See the region statement described in section 3.4.3.) The fifth line declares the name FOO to be a new data type, a subrange. Having declared FOO like this, the programmer can then write

```
declare ( X, Y: array FOO of boolean )
```

to declare each of X and Y to be vectors from 1 to 100 of booleans. The syntax used comes from lines 38, 88, and 95.

Note the distinction between the declaration of FOO above and the following declaration:

```
declare ( FUM: [1..100] )
```

This declares FUM to be an integer variable whose values are restricted to the range from 1 to 100, while FOO represents the type subrange from 1 to 100. This last example could alternatively have been written

```
declare ( FUM: FOO )
```

with equivalent effect. A possible use of FUM is

```
FUM := FUM + 1
```

to increment the integer FUM by one, with a possible check (depending on compiler options) if FUM goes out of its specified range.

---

between the <decl>s in this one, as required by the syntax of lines 12 and 32. A lexical artifice in the COL (explained in section 5.5.3) permits a semicolon to be omitted in most cases if it appears at the end of a line. This rule would permit the four semicolons to be omitted in this example. Subsequent examples use the rule and omit trailing semicolons on lines.



Note the following example:

```
declare ( Z is (A, B, C, D) )
```

This declares Z to be an enumeration type, for items that take on only the four values shown. (This feature is described in section 2.4.4.) The syntax then permits

```
declare ( Q array Z of integer )
```

which specifies that the subscripts of Q are those values. That is, Q[A] is permitted but Q[1] is not. Finally, one may then write an iteration statement such as

```
for N in Z do ... Q[N] ... endfor
```

to iterate N through the values of the enumeration type. (The iteration statement is described in section 3.3.2.)

Further examples may be found in Chapter 7.

## 2.6: Macro Declarations

The COL provides for string substitution macros, both without and with parameters.

```
127 <macro declaration> ::=
128     macro ID = <macro body>
129     macro ID ( <macro param> , ... ) = <macro body>

--- <macro body> ::=
    (see section 5.2)

130 <macro param> ::=                // formal parameter
131     ID
```

Line 128 provides a parameterless macro. Any instance of the declared identifier within the scope of this declaration is

replaced by the string which is its body. In the form on line 129 there is replacement of the formal <macro param>s in the parentheses with actual parameters supplied when the macro is used. All binding is done at the place of invocation.

The <macro body> is a special kind of quoted string in which the first non-blank character after the equal sign is used as the quote character. The body then continues up to the next instance of that character. Since the programmer may select as a quote character any character that he does not need to use in the body, there is no provision for including the quote character. See section 5.2 for further details.

The name of a macro is block structured as are other names in the COL. However, a special mechanism is required for there to be a hole in the scope of such a name, since otherwise the appearance of the name in the new declaration (or undeclaration) would be replaced by the macro text. The compiler directive "%literal" suppresses scanning for macro names on the rest of the line in which it appears. See section 5.3.4.

Further discussion of the COL's macro facility appears in section 6.1.2.3, along with some examples. The syntax for macro invocation is presented in section 5.2.

## 2.7: The Scope of Names

In general, name scoping in the COL is block structured as in ALGOL-60, with a few exceptions. First it is appropriate to define the term "scope" as it is used in this document.

DEFINITION: The scope of a declaration of a name is that portion of the input text in which the properties of that name are controlled by that declaration.

Note that it is the declaration that has a scope and not the name. Note further that the scope of a declaration is always deducible from static examination of the text of the program; flow of control at run time is not relevant. See also the discussion of this term in Appendix III.

In general, the scope of a declaration in the COL is from the beginning of the <declaration> in which the identifier is declared to the end of the smallest enclosing block (defined below), but the scope of a constant (as declared on line 52) starts at the point of declaration. In the text area just specified, the programmer has access to the variable with the properties given. The reason for the special rule for constant declarations is to permit such an ID to be redeclared later in the same declaration. An example using this ability is presented in section 6.1.2.3.

The term "block" used in the preceding paragraph (and also earlier in this document) refers to any COL context in which declarations may appear. Syntactically, this is any place in which "SD ; ..." is used in the syntax. For example, a routine

body is such a place, as indicated on line 15. Following is a complete list of all such places:

- . function body. The formal parameters of the function may be thought of as being declared at the beginning of this block. See section 2.1.
- . routine body. The previous comment about formal parameters applies. See section 2.1.
- . sequence, within "begin...end" or "{...} brackets". See section Ch3.
- . initialization and finalization code in a capsule. See section 6.1.3.
- . between "failing" and "failhere", and between "failhere" and "endfail". See section 3.4.4.
- . at the top level in a module. See section 6.2.2.
- . within "code" brackets. See section 6.3.1.

There are many places in the COL at which declarations may not appear. For example, the syntax for one form of conditional statement is

```
if E do S ; ... endif
```

If a declaration is required after "do", a nested block surrounded by "begin...end" or "{...}" must be used.

AD-A047 392

BOLT BERANEK AND NEWMAN INC CAMBRIDGE MASS  
COMMUNICATIONS ORIENTED LANGUAGE (COL): LANGUAGE DEFINITION.(U)  
MAY 77 A EVANS, C R MORGAN

F/G 9/2

DCA100-76-C-0051

NL

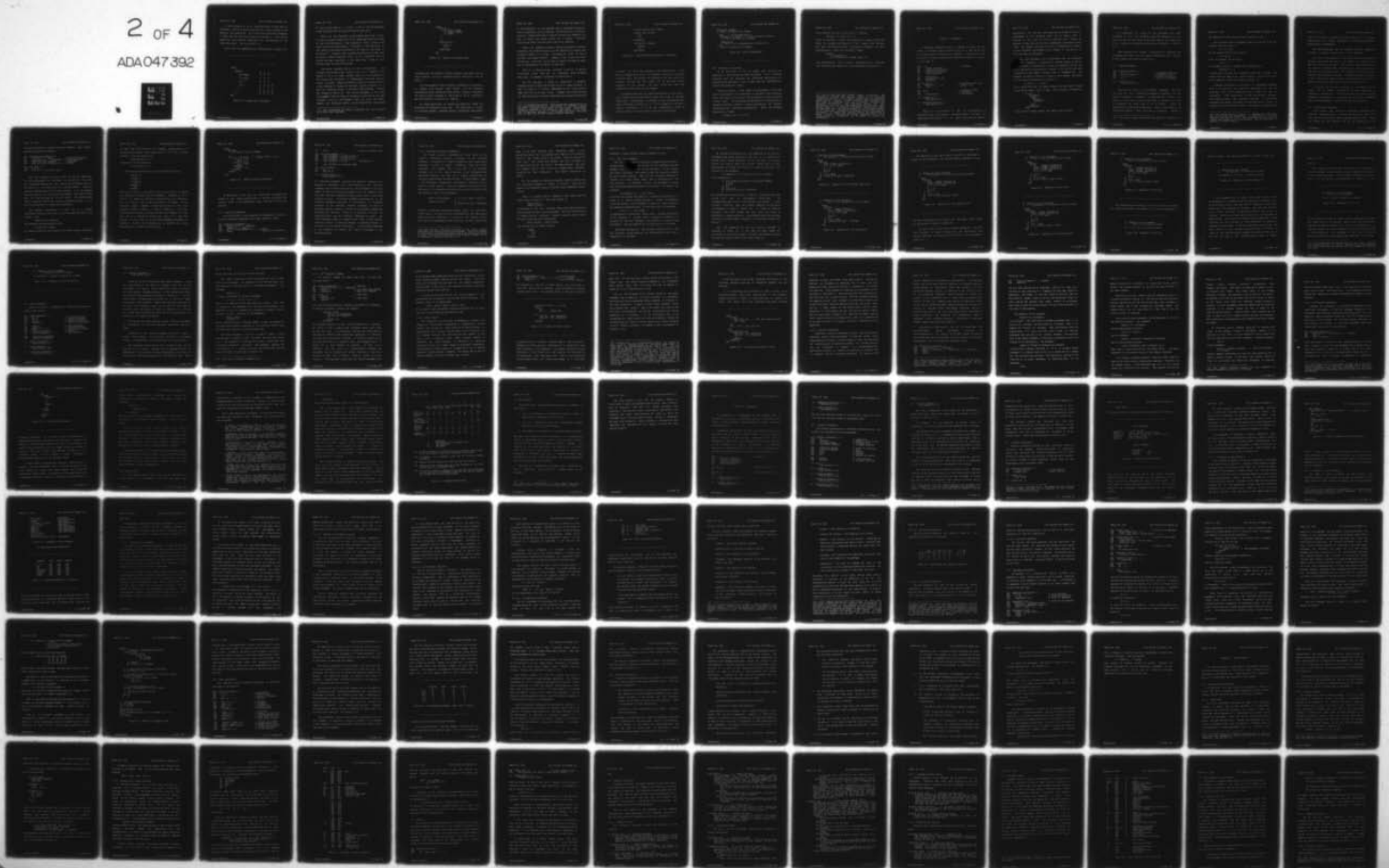
UNCLASSIFIED

BBN-3534

SBIE-AD-E100007

2 OF 4

ADA047392





A label constant is an ID appearing before a colon before a statement. Such an ID must be declared as a label constant in the smallest enclosing block. As in most block-structured languages, a "goto" may not lead into a block from outside it. This rule is implied by the preceding requirement for the placement of the label declaration. See also section 3.5.

These rules are suggested by the example shown in Figure 2-4.

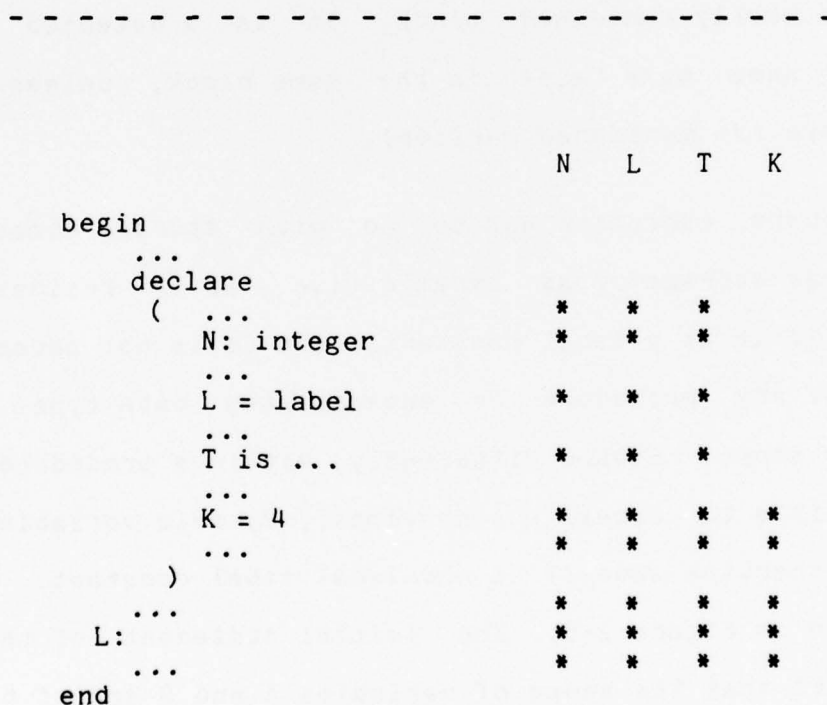


Figure 2-4: Scope Rule Illustrated

For each of the names N, L, T and K, a "\*" in the corresponding column indicates that the name is known at that point.

There are two exceptions to the general scope rule. First, if there is another declaration of the same name within the scope of the first declaration, that declaration forms a hole in the scope of the first declaration. That part of the text which is the scope of the second declaration is not part of the scope of the first(11). However, such a nested declaration of a name must be within a wholly contained block. It is a detected error to declare the same name twice in the same block, unless it is a constant name (as mentioned earlier).

The second exception has to do with storage class. If a variable has storage class dynamic (i.e., if it resides on the stack), or if it is a label constant, then it is not accessible in the body of any procedure or encapsulated data type declared within its scope. Stated differently, within a procedure body it is not possible to access any non-local, dynamic variable, or to goto (or otherwise access) a non-local label constant. Consider the example in Figure 2-5. The initial statement of the scope rule suggests that the scope of variables A and B and of the label constant L is all of this example (that is, up to the "end"). However, the limitation implied by the restrictions just stated

---

(11) This is precisely the effect in ALGOL-60 and PL/I, as well as in many other languages.

```
begin
  declare
    (  A: static integer
      B: dynamic integer
      L = label
    )
    ...
  L:
    ...
  routine FOO()
    ...
    point X
    ...
  endroutine
  ...
end
```

Figure 2-5: Scope in a Procedure Body

-----

indicates that the variable B (which resides on the stack) and the label constant L are not accessible within the body of FOO, at point X.

For the purposes of this discussion, parameters to procedures by default have storage class dynamic and are therefore inaccessible within nested procedure bodies. However, a parameter for which a static <storage> is specified is not so restricted. See the discussion in section 2.1, page 46.

It seems appropriate to explain the reason for these two exceptions to the general scope rule. The restriction on access to non-local dynamic variables permits a significant improvement

in the efficiency of the compiled code in procedure invocation, since a display(12) is not required. The restriction on access to non-local labels means that no special routines are needed at run time to adjust the stack pointer for a "goto". Thus both restrictions permit more efficient code to be compiled.

There is an apparent problem in declaring mutually recursive procedures, the problem arising from the necessity to declare each procedure before it is used. The syntax of lines 13 and 16 provides the needed facility. Suppose A and B call each other recursively. Then code such as that in Figure 2-6 might be used. The two declarations of B agree, as they must.

There is no special syntax provided in the COL to declare co-routines, since they can be programmed using existing facilities. An example is provided in section Ap2.2.

The COL provides a syntax to "undeclare" a variable. Although doing so causes the variable to be no longer declared, it does not cause it to revert immediately to any older value it might have had in an outer block. Instead, such reverting does not take place until the block containing the original declaration

---

(12) In implementing general block structured languages which do not impose this restriction, some method is required to access non-local variables which reside on the stack. This method, frequently referred to as a display, incurs costs at run time. The COL has been designed so as to avoid this cost. This good idea is taken from the BCPL design of Martin Richards.



```
forward routine B(N: integer)
routine A(W: boolean)
    ...
    B(2)
    ...
endroutine
routine B(N: integer)
    ...
    A(true)
    ...
endroutine
```

Figure 2-6: Declaring Mutually Recursive Procedures

-----

is exited. Any use of the variable in the remaining part of the block is flagged as an error. The example in Figure 2-7 may help to clarify this. The boolean "X" declared in the inner block is a different variable from the integer "X" in the outer block, occupying its own storage. The part of the text after the "undeclare" is the point to this example.

An undeclaration does not cause finalization to take place -- only normal exit from the block in which the item is declared causes that. (Finalization is relevant to only encapsulated data types, described in section 6.1.3.) If X in Figure 2-7 had been a capsule, finalization of it at the undeclaration would be incorrect since X is still known at the end of the figure.



```
declare(X: integer)
// Here X is known as an integer
...
begin // An enclosed block.
  declare (X: boolean) // Hole in scope of outer X.
  // Here X is known as a boolean.
  ...
  undeclare(X)
  // Here it is a detected error to refer to X.
end
// Here X is known as an integer.
```

Figure 2-7: Use of "undeclare"

- - - - -

## 2.8: Declaration Processing

It is the intent of the COL design that recursion be permitted in both program and data structures. It is therefore necessary that the algorithm for processing declarations be specified with precision, so as to insure that any implementation matches the designers' intent.

Note the problem: If the scope of a declaration is from the point of declaration of an identifier to the end of the containing block, then there is no way for two structures to contain pointers to each other. The following interpretation of a declaration is therefore required. Within a <scalar declaration> the constant declarations are interpreted sequentially. Hence the following declaration is legal:

```
declare ( N = 4; M = N+1 )
```

(This declares N to be 4 and M to be 5.) Writing

```
declare ( M := N+1; N := 4 )
```

would be incorrect (unless this declaration appears within the scope of a suitable declaration of "N"). However, the variable and type declarations within a declaration statement all occur simultaneously. Hence the following is legal:

```
declare  
  ( P is pointer Q  
    Q is structure(N: integer; NEXT: P)  
  )
```

This interpretation can be readily implemented by an algorithm that makes multiple passes over the declaration structure(13).

---

(13) Such an algorithm might operate roughly as follows: The declaration is passed over multiple times. On the first pass constant declarations are processed sequentially; on subsequent passes they are ignored. On each pass the type information for each item is recomputed from the information available from preceding declarations, from preceding passes over this declaration, and from the preceding information from this pass. Whenever there is no additional information needed after a pass, the algorithm has concluded successfully and processing ceases. If no additional information is obtained during a pass and more information is needed, the compiler reports the declaration as being in error. The algorithm always terminates since only a finite amount of information can be needed to process any declaration.

## Chapter 3: Statements

A statement, sometimes called a command, is obeyed for its effect. A statement DOES NOT HAVE A VALUE, so it may not be used in a context where an expression is expected. A discussion of the distinction between statements and expressions appears in section 1.1.2 on page 14.

```
132 S ::=                                // statement
133     <simple statement>
134     <conditional statement>
135     <iteration statement>
136     <other statement>

137     <assertion>
138     <machine-like code>
139     <sequence>

140 <sequence> ::=                        // statements and decls
141     begin SD ; ... end                // begin...end
142     { SD ; ... }                     // {...}

143 SD ::=                                // sequence element
144     S                                // statement
145     <declaration>                    // declaration

--- <assertion> ::=
    (see section 6.4)

--- <machine-like code> ::=
    (see section 6.3)
```

There is no distinction in the COL corresponding to ALGOL-60's block and compound. The marks "begin" and "end", or equivalently the braces "{ ... }", serve to delimit the scope of

declarations. For the most part they are not needed to serve as parentheses at the statement level (as do "begin...end" in ALGOL-60 or "do...end" in PL/I), since the COL's syntax is designed so that the need for such parentheses is obviated in most cases. For example, the body of an "if" is terminated by "endif", so that statement parentheses are not needed if the body is to hold more than one statement.

Note that statements and declarations may be intermixed freely in a sequence. As explained in greater detail in section 7, the scope of a declaration is from its appearance to the end of the smallest enclosing block. However, it is not the case that a declaration may appear anywhere that a statement may. For example, as shown in section 3.2 later in this chapter, the syntax for the one-arm conditional in the COL is

```
if E do S ; ... endif
```

Thus a declaration may not appear between "do" and "endif" unless an enclosing "begin...end" is used. The following is acceptable:

```
if E do
  statements
  begin
    declare ...
    statements
    ...
  end
  statements
endif
```

This would be illegal without the "begin" and the "end".



An <assertion> is a claim by the programmer that some predicate is true. Its presence helps the readability of the code, and the user may choose to have the compiler take cognizance of it. Details of the options available are presented in section 6.4.

Where necessary for reasons of efficiency or otherwise, the programmer may insert into his program machine-like code. Details of this feature are given in section 6.3.

### 3.1: Simple Statements

```
146 <simple statement> ::=
147     E := E                // assignment statement
148     E *= <infix-op> E     // alternate form
149     E ( E , ... )        // routine invocation
150     E ( )                // ditto, no arguments

--- <infix-op> ::=
    (see section 4.2)
```

Note the two forms of the assignment statement. The first one (line 147) is a conventional assignment statement. The left side is evaluated in L-mode(14) to determine the location into which to do the store, and the right side is evaluated in R-mode to determine the value that is to be stored. (These two evaluations are performed in whichever order is found best by the optimizing compiler.) Then the store is performed. Details of

---

(14) The terms "L-mode" and "R-mode" are defined in Appendix III.



the semantics of the store are found in section 4.2.2.1.

The alternate form of assignment shown on line 148 is for the commonly occurring case of

$$E1 := E1 \text{ op } E2$$

The COL permits this to be written as

$$E1 \text{ } *= \text{ op } E2$$

Thus, for example, one may write

$$X \text{ } *= \text{ } + \text{ } 1$$

to increment X by one. Somewhat more interesting is

$$A[3*n+2] \text{ } *= \text{ } + \text{ } 1$$

in which more writing and (hopefully) more code is saved. The class <infix-op> used in the syntax refers to any of COL's infix binary operators; these are listed in section 4.2.

Although the syntax for both kinds of assignment statement shows E on the left side, obviously some types of E are not permitted. Only those expressions may be used that have an L-value(15). Such expressions include the obvious: variable, subscripted variable, and structure reference. A conditional expression may appear, providing that each arm has an L-value. An expression of the form "E @" (where E evaluates to a pointer) may

---

(15) The phrase "has an L-value" is somewhat less restrictive than is the phrase "is addressable". In particular, a structure component that does not lie on word boundaries is said to have an L-value even though it is not addressable. See the discussion in Appendix III under "memory".

appear; the semantics is that the location pointed to is updated. (See section 4.4.) A macro may appear if the result after macro substitution is acceptable.

Any value whatsoever may be assigned across an assignment statement, including in particular an entire aggregate.

The type of the expression on the right side of an assignment statement must match the type of the variable (or component) into which the store is to be done. This check is always performed by the compiler at compile time, consistent with the design decision that the COL be strongly typed. The compiler will in certain very limited cases provide an automatic conversion (called a coercion) of the value on the right to make it match the type on the left. Details about these coercions are given in section 6.1.5.

A routine call is the invocation of a procedure that is called for its effect. Declaration of such procedures is presented in section 2.1. In an invocation of a routine, actual parameters are supplied to correspond to the formal parameters used in the declaration. Then the routine body is obeyed.

### 3.2: Conditional Statements

Both one-arm and multi-arm conditional statements are provided. Separate syntax is used for the two forms, since it is felt that this distinction makes programs easier to read. (Both the reader and the compiler know at the beginning of the

construction whether to expect one arm or several.) Also, certain syntactic problems are avoided.

```
151 <conditional statement> ::=
152     if E do S ; ... endif           // one arm conditional
153     unless E do S ; ... endunless  // alternate form
154     test <alt list> endtest         // multi-arm
155     test <alt list> otherwise S ; ... endtest

156 <alt list> ::=
157     E do S ; ...
158     E do S ; ... orif <alt list>
```

The "if" construct of line 152 has the obvious semantics. First the expression E is evaluated. If its value is true, then the statements between "do" and "endif" are performed; while if the value of E is false there is no further effect. Since the compiler requires that E be of type boolean, it is not possible that the value of E at run time be non-boolean. Note that the COL uses "if...do" and does not use "if...then". The reserved word "then" is used in the COL for a totally different purpose. (It is part of the syntax of a conditional expression.)

The "unless" construction of line 153 is a one-arm conditional similar to "if", but it uses the opposite sense of the boolean. Thus

```
unless E do S endunless
```

has precisely the same effect as does

```
if not(E) do S endif
```

In certain situations the "unless" form is more natural and easier

to read than the "if" form. For example, suppose Error is a parameterless function that returns true if an error has been detected. Then the construction

```
unless Error() do ...
```

seems easier to read than does

```
if not Error() do ...
```

The multi-arm conditional of line 154 looks like this:

```
test E1 do
  S1
orif E2 do
  S2
orif E3 do
  S3
otherwise
  S4
endtest
```

Only one of the  $S_k$  is executed. If  $E_1$  is true then  $S_1$  is obeyed and execution continues after the "endtest"; otherwise if  $E_2$  is true  $S_2$  is obeyed and execution continues after the "endtest"; etc. If none of the  $E$ 's are true, then  $S_4$  is obeyed. If the otherwise clause is absent, it may be that none of the  $S_k$  is executed. Another example is shown in Figure 3-1. This code initializes the X-array, setting the first five elements to 0, the next ten to 1 through 10, and the last five to 10. Each time through the loop only one of the three assignment statements is executed. (A much more efficient way to perform this array initialization is shown in section 4.3.)

```

begin
  declare
    ( X: array [1..20] of integer
    )
    ...
  for N := 1 to 20 do    // Iterate N from 1 to 20
    . st N <= 5 do
      X[N] := 0
    orif N <= 15 do
      X[N] := N-5
    otherwise
      X[N] := 10
    endtest
  endfor
  ...
end

```

Figure 3-1: Sample Conditional Statement

-----

An implication of the way the conditional statements are defined is that labels placed within an arm are accessible from outside. There is no problem here, since the flow of control is explicit.

### 3.3: Iteration Statements

The COL provides a rich selection of statements to execute a group of statements repeatedly until some condition is met.

```

159 <iteration statement> ::=
160   while E do S ; ... endwhile
161   repeat S ; ... until E
162   for <for list element> do S ; ... endfor
163   break                               // skip out of iteration

```



```
164      loop                                // skip to iteration test
165  <for list element> ::=
166      <for variable> := E step E until E
167      <for variable> := E incr E to E
168      <for variable> := E decr E to E
169      <for variable> := E to E              // increment by 1
170      <for variable> in <discrete type>
171  <for variable> ::=
172      ID
173      defined ID
---  <discrete type> ::=
      (see section 2.4.4)
```

An <iteration statement> provides for repetitive execution of a sequence of statements until some condition is met. The "for" statement uses a counter specified by the programmer to count the number of times the controlled statement is to be repeated, while the other <iteration statement>s repeat either while or until some condition is true. The "break" statement terminates any of the iteration statements described in this section. It terminates execution of the innermost iteration statement in which it textually appears, causing a transfer to the statement immediately following the iteration. (There is no provision in the COL for terminating more than one iteration statement at a time.) The "loop" statement causes a transfer to the test of the innermost iteration in which it textually appears to determine whether or not there is to be another replication. A more precise statement of the semantics of "break" and "loop" is provided in the discussion below.

### 3.3.1: The Simple Iteration Statements

The simple iteration statements are those which do not involve a controlled variable introduced by the iteration statement. The semantics of these statements is most conveniently described in terms of simpler statements, using the "goto" statement. The examples show on the left a COL iteration statement and on the right a sequence of COL statements with equivalent semantics but made up of simpler constructions (or iteration constructions already defined). In each example, labels L and B designate respectively the places to which any loop or break in S cause transfer. They are assumed to be declared at the beginning of the smallest enclosing block and to be distinct from any identifier used by the programmer.

while E do S endwhile	L: if E do S; goto L endif; B:
repeat S until E	P: S
	L: unless E do goto P endunless
	B:

Note(16) an important distinction between these two constructs. In the "while...do" form the first performance of the test occurs before the first execution of S, while in the "repeat...until" form the first performance of the test does not take place until

---

(16) Note that the semicolons required by the syntax between statements are omitted in the second fragment. This is an example of the lexical rule permitting semicolons to be omitted at the end of lines. This rule was mentioned in the footnote on page 77; it is explained in detail in section 5.5.3.

after S has been executed once. Therefore there is the possibility that S will be executed zero times (if E is initially false) in the former, while in the latter S must be executed at least once. The latter form permits the test to depend on variables that are not set until S has been executed. Note that the "repeat...until" form is ever so slightly more efficient, requiring one less instruction. (One "goto" instruction is saved.)

The COL imposes no restriction against a "goto" leading into the controlled statement of "while" or "until". Note that the semantics as presented leaves no question about subsequent flow of control.

There is a certain lack of symmetry in this syntax that the reader should be aware of. Note that neither of

```
until E do S
repeat S while E
```

is permitted in the COL. Although these forms seem attractive, they introduce serious parsing problems. Note also that there is no "repeat forever" construct. However, one may define a macro "forever" using the macro declaration

```
macro forever = "until false"
```

(see section 2.6) to permit writing

```
repeat
  S1
  S2
  ...
forever
```

Presumably a break statement appears somewhere in this.

### 3.3.2: The "for" statement

The "for" statements are iteration statements which specify a variable which is initialized as part of the statement and then changed. Note the forms of <for variable>. In the simpler form without "defined", the intent is that the iteration variable be local to the iteration. That is, the iteration statement is replaced by a block which starts with a declaration of the <for variable>. If "defined" is used, the programmer is saying that an already-declared variable is to be used. For example, the statement

```
for defined V := ... do S endfor
```

is correct only within the scope of a suitable declaration for V. (That is, it uses an already defined V.) Further, on completion of the execution of this statement, V holds the value implied by the semantics presented below. On the other hand, the statement

```
for V := ... do S endfor
```

is equivalent to a block that starts with a suitable declaration of the variable V. As this block is exited on completion of execution of the "for" statement, the variable V no longer exists thereafter. Thus the question of V's final value is irrelevant.

Note that the syntax of <for variable> requires an ID. Thus the iteration variable may not be subscripted nor may it be a component of a structure.



We proceed by defining first the semantics of the iteration statement whose syntax is given on line 166. It is convenient to present first a simple definition of this form that is not quite correct, and then to repair the problem. Using B and L to label destinations for "break" and "loop" in S as before, we define the semantics of the statement

for defined V := E1 step E2 until E3 do S endfor

to be equivalent to

```
// This simple equivalence is NOT QUITE CORRECT.
  V := E1
  goto Q
P:  S
L:  V *= + E2
Q:  unless E3 do goto P endunless
B:
```

The problem with this definition is that E2 is evaluated each time through the loop, an unacceptable inefficiency. The straightforward way to solve this problem is to declare a local variable to hold it. Thus the correct semantics for the above for-statement is shown in Figure 3-2. Of course E3 must be evaluated each time through the loop, since its value is presumably to change eventually. This example shows Step as being declared to be type integer; actually, it is declared to match the type of V.

Next, the semantics for the form without "defined" is presented in Figure 3-3. V and Step are again shown for convenience to be of type integer; actually, the compiler selects the type that matches that of the initial value E1.



```
// Semantics of the statement
//   for defined V := E1 step E2 until E3 do S endfor

begin
  declare
    ( Step: integer initially E2
      B, L, P, Q = label
    )
    V := E1
    goto Q
  P: S
  L: V *= + Step
  Q: unless E3 do goto P endunless
  B:
end
```

Figure 3-2: Semantics of "for defined step until"

-----

```
// Semantics of the statement
//   for V := E1 step E2 until E3 do S endfor

begin
  declare
    ( V: integer initially E1
      Step: integer initially E2
      B, L, P, Q = label
    )
    goto Q
  P: S
  L: V *= + Step
  Q: unless E3 do goto P endunless
  B:
end
```

Figure 3-3: Semantics of "for step until".

The semantics of the "incr" form of line 167 is presented in Figure 3-4 with "defined". For the form without "defined", we use

```

- - - - -

// Semantics of the statement
//   for defined V := E1 incr E2 to E3 do S endfor

begin
  declare
    ( Step: integer initially E2
      Final: integer initially E3
      B, L, P, Q = label
    )
    V := E1
    goto Q
  P: S
  L: V := + Step
  Q: if V <= Final do goto P endif
  B:
end

```

Figure 3-4: Semantics of "for defined incr"

the same transformation shown above for the "step...until" form; the equivalence is shown in Figure 3-5.

The "decr" form is similar but iterates downwards. The form with "defined" is shown in Figure 3-6, and the similar form without "defined" is treated as for the "incr" case, as shown in Figure 3-7.

```
// Semantics of the statement
//   for v := E1 incr E2 to E3 do S endfor

begin
  declare
    ( V:      integer initially E1
      Step: integer initially E2
      Final: integer initially E3
      B, L, P, Q = label
    )
    goto Q
  P: S
  L: V *= + Step
  Q if V <= Final do goto P endif
  B:
end
```

Figure 3-5: Semantics of "for incr"

- - - - -

```
// Semantics of the statement
//   for defined V := E1 decr E2 to E3 do S endfor

begin
  declare
    ( Step: integer initially E2
      Final: integer initially E3
      B, L, P, Q = label
    )
    V := E1
    goto Q
  P: S
  L: V *= - Step
  Q: if V >= Final do goto P endif
  B:
end
```

Figure 3-6: Semantics of "for defined decr"

```

// Semantics of the statement
//   for V := E1 decr E2 to E3 do S endfor

begin
  declare
    ( V:      integer initially E1
      Step: integer initially E2
      Final: integer initially E3
      B, L, P, Q = label
    )
    goto Q
  P: S
  L: V *= - Step
  Q: if V >= Final do goto P endif
  B:
end

```

Figure 3-7: Semantics of "for decr"

-----

The increment may be omitted if it is one, as shown in Figure 3-8. Here we have shown the semantics of one <for-statement> in

-----

```

// Semantics of the statement
//   for V := E1 to E3 do S endfor

for V := E1 incr 1 to E3 do S endfor

```

Figure 3-8: Semantics of "for to"

terms of another. The form with "defined" is shown in Figure 3-9.

```
-----  
  
// Semantics of the statement  
//   for defined V := E1 to E3 do S endfor  
  
for defined V := E1 incr 1 to E3 do S endfor
```

Figure 3-9: Semantics of "for defined to"

```
-----
```

A few comments are in order. Note that in all of the for-statements it is possible that the controlled statement will not be executed at all, since the test for completion is performed before the first execution of the body. Note also that the test in the "incr" and "decr" forms works correctly only if E2 is positive. (There is no check made unless E2 can be evaluated at compile time. It can easily be checked using an assertion.) Note also that the COL provides no exact equivalent to ALGOL-60's step-until construction. In ALGOL-60 the upper limit follows the until, while in COL a boolean follows the until. The forms on lines 167 and 168 more closely resemble the ALGOL-60 form, but in the COL the sign of the increment must be known at compile



time(17).

The iteration form involving "in" on line 170 provides for iterating through a discrete type, using an integer or enumeration as the iteration variable. The form without "defined" is shown in Figure 3-10 and the form with "defined" in Figure 3-11. (The

-----

```
// Semantics of the statement
//   for V in R do S endfor

for V := low(R) to high(R) do S endfor
```

Figure 3-10: Semantics of "for in"

-----

built-in functions "low" and "high" return respectively the lower and upper limit of the discrete type to which they are applied; they are described in section 4.4.) If R is an enumeration, V is iterated through all of its elements in the order they are listed in the declaration. The advantage of this form is that the iteration through the array matches the declaration of the array.

---

(17) This design decision permits the COL to avoid a serious inefficiency inherent in a correct implementation of the ALGOL-60 form, since that language requires that the increment be evaluated each time.

```
// Semantics of the statement
//      for defined V in R do S endfor

for defined V := low(R) to high(R) do S endfor
```

Figure 3-11: Semantics of "for defined in"

-----

### 3.4: Other Statements

Those of the COL's statements not previously presented are given in this section.

```
174 <other statement> ::=
175     goto ID                      // the notorious goto
176     ID : S                      // labeled statement
177     return                      // return from routine
178     resultis E                  // function value return

179     <empty>                    // empty statement
180     swap ( E , ... )           // interchange items
181     free ( E , ... )          // return to free store
182     <switch statements>        // switchon
183     <Zahn statements>         // Zahn's device

184     <interlock statements>
185     <failure statements>

--- <switch statements> ::=
    (see section 3.4.1)

--- <Zahn statements> ::=
    (see section 3.4.2)

--- <interlock statements> ::=
    (see section 3.4.3)
```

--- <failure statements> ::=  
      (see section 3.4.4)

Lines 175 and 176 provide the usual "goto" statement. There are restrictions on the part of a program which may be the destination of a "goto". Many of these are stated in connection with the relevant language construct in this document and others are implied by the discussion of scope in section 2.7. All such restrictions are summarized in section 3.5, to which the reader should give close attention. Recall also that each identifier to be used as a statement label as in line 176 must first be declared, using the syntax on line 53 of section 2.2.2. Any use of a label must be within the scope of such a declaration. See section 3.5 for further discussion.

The COL does not permit passing a label value as a parameter to a procedure, nor are there variables whose value may be a label.

Execution of the "return" statement on line 177 causes a return from a routine. It is an error to use it in any other context. In particular, it may not be used in a function body.

The "resultis" construction of line 178 may be used only in the body of a function. Obeying such a statement causes the expression E to be evaluated and used immediately as the value of the function, terminating execution of the function body. There

may be more than one "resultis" within the body.

The "swap" construct of line 180 provides for cyclic right shift of its arguments. All arguments must be the same type, but that type may be any type at all (including an aggregate). The statement

```
swap(A, B, C)
```

is roughly equivalent to the four statements

```
T := C; C := B; B := A; A := T
```

where T is a temporary of the same type as A, B and C. Note that if "swap" has two arguments, they are merely interchanged. The semantics is that all arguments are evaluated in L-mode(18) before any store is done. Thus each of the statements

```
swap(K, N[K])  
swap(N[K], K)
```

has an identical effect. Although "swap" is easy to program for any particular instance, providing it as a built-in routine helps the optimizing compiler to produce better code.

The "free" statement (line 181) is used to return to free storage space obtained by the "allocate" function (see section 4.3). The arguments to free must be pointer values that were originally returned by "allocate". This statement compiles into an invocation of a variadic run-time subroutine that does the work; the latter is described in section 6.5.

---

(18) This term is defined in Appendix III.

## 3.4.1: The "switchon" Command

The switchon command is taken from BCPL; it uses the following syntax:

```
186 <switch statements> ::=           // switchon
187     switchon E into S ; ... endswitch
188     <case> : S                     // case label (for switch)
189     stopswitch                     // skip out of switch

190 <case> ::=                         // case label
191     case NC
192     case NC to NC                 // many cases
193     default
```

Note that the syntax permits multiple case labels on a statement.

Proceeding by example, consider the fragment:

```
switchon E into
    case N1: S1; stopswitch
    case N2: S2; stopswitch
    ...
    default: Sn
endswitch
```

The intended semantics is that first expression E is evaluated, and there is then a jump to that statement whose case label matches that value. It is important that the case labels MUST BE EVALUABLE AT COMPILE TIME, permitting generation of efficient code, and that they must be distinct. If E does not match any of the cases and there is a "default" label, the statement which it labels (Sn in this example) is executed. If there is no "default", then the action is unpredictable, although the programmer may be confident that either some one of the Sk is executed or that none of them is. (That is, there is no wild transfer.) The programmer should omit the default statement only



if the program logic absolutely precludes the possibility of the value failing to match some one of the case labels. Note that each  $S_k$  falls through into the next one. The command "stopswitch" causes a jump to the end of the smallest enclosing switch and frequently appears after each  $S_k$ .

The "case N to N" form on line 192 may be thought of as an abbreviation for writing out all the case labels indicated. For example, "case 1 to 5" abbreviates

case 1: case 2: case 3: case 4: case 5

It is possible that the compiler produces different code for these two ways of expressing the same semantic idea.

#### 3.4.2: Zahn's Device

The literature on structured programming is replete with examples of the problems encountered in attempting to program without goto, as well as clever solutions to these problems. In designing the COL a careful study was made of this literature, and one proposed solution has been selected that appears to have merit, that of Charles T. Zahn. Zahn's proposal appears in [Zahn-74], and Knuth presents in [Knuth-74] an excellent discussion of this and other issues in structured programming. The discussion of Zahn's device is on pages 275 to 277 of Knuth's paper and includes several examples. The syntax used in the COL differs slightly from that of either Zahn or Knuth.

```

194 <Zahn statements> ::=                // Zahn's device
195     upon ID or ... leave S ; ... do S ; ... endupon
196     signal ID                        // part of Zahn's device

```

The construct on line 195 is Zahn's device, and the syntax on lines 188, 189 and 196 is relevant. Again proceeding by example, consider Figure 3-12. The intended effect is that each of the IDk

```

- - - - -
    upon ID1 or ID2 or ... or IDn
    leave
        S11
        S12 ..... signal IDk
        ...
    do
        case ID1: S21; stopswitch
        case ID2: S22; stopswitch
        ...
    endupon

```

Figure 3-12: Example of Zahn's Device

is declared to be a distinct constant name of type "condition", somewhat as if they were the components of an enumeration, with scope extending to the "endupon". The command "signal ID" causes termination of the "leave" part and a jump as for "switchon" to the appropriate case label after "do". Each of the identifiers listed between "upon" and "leave" must appear as a case label

after "do". In the event that control "falls off the end" of the "leave" part, the effect depends on whether or not a "default" label appears after "do". If it does, the code so labeled is jumped to; it is a detected error if not.

Any of the IDk named in the initial phrase of the "upon" statement may be passed as an actual parameter to a procedure, providing that the corresponding formal parameter is declared to have type "condition". A signal of that formal parameter within the procedure body causes a jump to the proper case label on the call side. This mechanism can provide a convenient error return mechanism for a procedure. It is anticipated that it be implemented by a "skip return"(19) from the procedure. Within the procedure body the condition may be passed as an actual parameter to still another procedure. An example of this is presented in section 6.1.4.2.

---

(19) Although this is an implementation detail and beyond the usual scope of language design, it nonetheless seems useful to record here the implementation that we have in mind. Call the normal return address of a subroutine RA. Then a procedure with (say) two parameters of type condition will normally return to RA+2, returning to RA if the first parameter is signaled and to RA+1 if the second is signaled. On the call side, the compiled code will contain in RA and in RA+1 jumps to appropriate code to implement the signal. This implementation provides in the COL access to a coding device frequently used by assembler coders for efficient implementation of error returns from procedures. Of course any implementor is free to choose another implementation if appropriate.

Since the scope of the signal constants IDk extends to the following "endupon", they may be "signal"ed between "do" and "endupon".

It is interesting that the effect of Zahn's device can be simulated (almost) using other constructions in the language. Compare the form in Figure 3-12 above with that in Figure 3-13 below. The latter form using "switchon" has almost the same

```
-----  
  
begin  
  declare  
    ( FOO: (ID1, ID2, ..., IDk, UND) initially UND  
      OUT = label  
    )  
  S11  
  S12 ... FOO := IDk; goto OUT  
  ...  
OUT:  
  switchon FOO into  
    case ID1: S21; stopswitch  
    case ID2: S22; stopswitch  
    ...  
    default:  
  endswitch  
end
```

Figure 3-13: Equivalence for Zahn's Device

semantics as does the former using Zahn's device. The IDk are declared as constants by declaring FOO to have the type enumeration class, and one of the IDk is assigned to FOO before the label OUT. (Each "signal" is replaced by an assignment to FOO followed by "goto OUT".) The branching after OUT corresponds to that after the "do" in Zahn's device. The only difference between the semantics of the code in Figure 3-12 and that in Figure 3-13 is that the former does not define a block and the latter does. That is, a label placed in the code of Zahn's device can be "goto"-ed to from outside, but the COL does not permit a "goto" into the block shown in Figure 3-13. Omitting the "begin" and the "end" does not solve the problem either, since it is permissible to use Zahn's device in a context in which a declaration is not permitted.

#### 3.4.3: Interlock Statements

Interlocks provide a mechanism to insure that two processes executing within a multi-processing or interrupt system cannot simultaneously be within a critical region of code, thus providing for synchronization of parallel processes. For further details, see the discussion in Chapter 3 of [Brinch-Hansen-73], particularly section 3.3. Briefly, an interlock is a special kind of variable with the following properties. It has one of two



values, referred to as "locked" and "unlocked"(20). The important thing that can be done with an interlock is a special operation called locking it. This action locks the variable if it was previously unlocked, and has no effect if it was already locked. It is of paramount importance that the actions of testing and setting in this instruction are a single atomic uninterruptible entity, so that, for example, there can be no intervening access to the lock by another processor between testing the interlock and setting it. Because of this requirement, interlocks must be a built-in data type in the COL, since they cannot be simulated by other constructs. The mechanism for implementing them is dependent on the hardware available. (This point is addressed further towards the end of this section.)

The pattern of interlocks in the COL is modeled after Per Brinch-Hansen's region statements, described in [Brinch-Hansen-72]. The idea is to provide a syntax for dealing with interlocks that is consistent with the dictates of structured programming. The COL provides the following syntax:

```
197 <interlock statements> ::=
198     region E do S ; ... endregion
199     region E iflocked S ; ... otherwise S ; ... endregion
200     retry
201     lock E
```

---

(20) These are primitive values somewhat like "true" and "false", with no other properties to their representation. They need not have well defined values, though. For example, in many implementations unlocked might be any non-zero value.

```
202    lock E iflocked S ; ... endlock
203    unlock E
```

The E in each case must be an interlock. (That is, it must be a variable of type interlock or a component of that type within an aggregate.) The intent is that the forms on lines 201, 202 and 203 should be needed rarely if at all, the structured forms on lines 198 and 199 meeting most needs. Further, it is intended that locks be used for access control of a very short duration, on the order of milliseconds.

The semantics of the statement

region E do S endregion

is as follows: The interlock E is tested repeatedly until it is found to be unlocked, at which point it is locked. Then S is obeyed, and finally E is unlocked. Note particularly that the program waits in a loop until the lock is clear, so that the programmer must use care in his locking strategy. The alternate form of the region statement on line 199 is available in case this looping is not satisfactory. The statement

region E iflocked S1 otherwise S2 endregion

involves first testing interlock E. If it is already locked statement S1 is obeyed, while if not it is locked and S2 is obeyed and the interlock then unlocked. The testing and setting of the interlock is an atomic operation, as described above. If the statement

retry

appears inside of S1, its effect is to hop back to the testing action. The compiler detects as an error the use of "retry" in any other context.

The form on line 201 causes a wait (if necessary) until E is unlocked followed by locking it. The form on line 202 involves a testing operation as for the form on line 199, with statement S being obeyed only if the interlock is previously locked. (The command "retry" may be used within S.) The form on line 203 serves to clear an interlock.

Note that the region statement can be explained in terms of the simpler statements: The statement

region E do S endregion

has the same semantics as does

lock E; S; unlock E

and the statement

region E iflocked S1 otherwise S2 endregion

has the same semantics as does

lock E iflocked S1; goto P endlock; S2; P:

Note that a retry within S1 has the proper effect. The region statement is a structured version of the simpler statements.

It is almost invariably important that the locked region of code not be exited without clearing the interlock. Therefore, any COL command causing a non-returnable jump out of this region causes the interlock to be unlocked. (The commands involved are

"break", "loop", "return", "resultis", "stopswitch", Zahn "signal", and "goto". This matter is discussed in detail in 3.5.) A procedure call within the locked region is not treated specially, although the programmer must be aware of the perils. An abnormal exit from the procedure that did not result in return to the region could leave the interlock locked, probably a disaster. A "fail" statement within the region does not cause the interlock to be cleared, unless the lock is named in the "fail finishing" option. A "goto" within code brackets can be used to jump out of the region without clearing the interlock. This construction is potentially quite dangerous and should seldom be needed.

An interlock may be compared using the "=" operator with either of the constants "locked" or "unlocked". However, it is important to note that it is not possible to write safely code with the same effect as the "lock" statement. Consider replacing

lock V

by the statements

```
while V = locked do endwhile // wait till unlocked
V := locked
```

This(21) appears superficially to have the same semantics, but it admits a serious hazard. In the short time interval after V is found to be unlocked and before the assignment is executed,

---

(21) The <empty> statement between "do" and "endwhile" is executed repeatedly as long as V is locked.



another processor might also lock V. Then each of two processes would think it owned the resource(22). It is imperative that the testing and setting be a single uninterruptible operation in the hardware.

#### 3.4.4: The "failure" Mechanism

Although the principles of structured programming provide significant advantages to both the language designer and the programmer, the discipline frequently deals poorly with the code needed to handle unexpected situations. The COL provides several solutions to this problem. The "failure" feature discussed in this section is intended to help the programmer deal with catastrophic failures in his application.

```
204 <failure statements> ::=
205     failing SD ; ... failhere SD ; ... endfail
206     fail
207     fail finishing E , ...
```

Note the example of the use of this feature outlined in Figure 3-14. Execution of the "failing" statement sets a global trap, so that any subsequent execution of either form of the "fail" statement (the distinction is explained later) causes the trap to be cleared and then a transfer to statement S11 immediately

---

(22) The probability of such an event is quite high if that other processor happens to be waiting in the same sort of loop on V. Clever programming can reduce the probability, but no simple software mechanism is completely safe in the absence of help from the hardware.



```
S0
failing
  declare ( ... )
  ...
  S1
  ...
  WORK(...)
  ...
  S9
failhere
  S11
  ...
endfail
S20
```

Figure 3-14: The "failure" Mechanism

-----

following "failhere". If no "fail" occurs and "failhere" is reached normally after execution of S9, the trap is cleared and execution continues after "endfail" at S20. It is important to realize that any executed "fail" is caught in this way, whether it occurs in the text between "failing" and "endfail" or in a procedure called. In this example, a "fail" occurring in procedure WORK would trap to the "endfail" shown.

A "fail" causes an abnormal exit from the place at which it appears. The "fail finishing" form on line 207 provides for the usual cleanup of a normal exit where needed. Each expression listed must denote either an interlock or a variable which is an encapsulated data type. They are finalized as they would be in a

normal exit. Specifically, interlocks are cleared and finalization is performed for capsules. For example, consider the following:

```
region L do
    ...
    fail finishing L
    ...
endregion
```

Execution of the "fail finishing" statement causes interlock L to be unlocked. Had "finishing" been omitted, L would be left locked.

Although only one failure trap can exist in the running program at any moment, the trap may be dynamically stacked. That is, "failing" saves the current trap, which is restored after "endfail". Section 6.1.4 presents a proposed implementation of the failure mechanism, along with discussion comparing this mechanism with the COL's other exception handling features.

A failure occurring in a multi-processing environment impacts only on the process in which it occurs. That is, each process has the responsibility of catching its own failure traps. Of course the programmer is free to write failure processing code that stops other processes if that is appropriate.

### 3.5: Restrictions

The COL imposes certain restrictions on the interactions of various features of the language. For the most part, each

restriction is discussed in this document in connection with the language feature to which it pertains. However, it seems useful to discuss all such restrictions in one place so as to make more clear the underlying philosophy that leads to them.

First some terminology is defined. To the extent that these are terms that are defined also in Appendix III, the definitions are consistent.

- . A BLOCK is any context in the COL in which declarations may appear. Syntactically, it is any context in which "SD ; ..." appears in the syntax. There is a complete listing of such contexts in section 2.7 on page 82.
- . A PROCEDURE BODY is the body of a procedure, either a function or a routine; the terms FUNCTION BODY and ROUTINE BODY have the obvious meanings. Note that each of these is also a block.
- . A REGION BODY is the part of a region statement in which the interlock is set. In the simple form without "otherwise" it is the code between "do" and "endregion"; in the other form it is the code between "otherwise" and "endregion". A region body is not a block.
- . A FOR-BODY is the controlled part of a for-statement, between "do" and "endfor". Although a for-body is not itself a block (in that the relevant syntax involves S rather than SD), it is block-like in the sense that it is the scope of the variables declared as part of the semantic definitions given in section 3.3.2.
- . A NORMAL EXIT from a block is a departure from the block in some controlled way that the compiler knows about. An ABNORMAL EXIT is either one that the compiler does not know about, or a "fail" statement, or any exit that occurs within code brackets.
- . Certain CLEAN UP actions are performed for the user on normal exit from a block. Specifically, if the block is a region body then the interlock is cleared on normal exit, and if any variables are declared of type capsule with a finalization specified then that finalization is

performed.

More details about some of these are presented below.

Most of the restrictions of this section are presented in summary form in Table 3-1. Each column concerns some COL construct that transfers control from one place to another, and each row concerns some context in the COL to or from which such a transfer may or may not take place. An entry of "-" means that the transfer is permitted and is a normal exit, an entry of "x" means that the exit is permitted but is abnormal, an entry of "no" means the transfer is not permitted, and an integer in [...] concerns a note at the bottom of the Table. Column 0 concerns a "goto" leading INTO the context, while the other columns concern some kind of transfer leading OUT OF a context. In particular, column 2 concerns a "goto" leading out, column 3 a "return" or "resultis", column 4 a "stopswitch", column 5 a "signal", and column 7 a "fail" or "fail finishing". Column 8 concerns falling off the bottom of a context.

The contexts defined by the rows are fairly obvious. The first four are defined at the beginning of this section. A switch body is that code between "into" and "endswitch". The two lines for Zahn's device refer respectively to that code between "leave" and "do", and that between "do" and "endupon". The next two lines refer to initialization and finalization in a declaration of an encapsulated data type. The final line refers

	0	1	2	3	4	5	6	7	8
	goto (in)	break loop	goto (out)	retrn rslts	stpsw	sgnal	retry	fail [5]	fall off
block	no	-	-	-	-	-	-	x	-
proc body	no	no	no	[1]	no	[2]	no	x	[3]
region	no	-	-	-	-	-	no	x	-
for-body	no	-	-	-	-	-	-	x	-
switch body	-	-	-	-	-	-	-	x	-
Zahn-leave	no	-	-	-	-	-	-	x	[4]
Zahn-do	-	-	-	-	-	-	-	x	-
failing	no	-	-	-	-	-	-	x	-
start	no	no	no	no	no	no	no	x	-
finish	no	no	no	no	no	no	no	x	-
code brkts	-	-	-	-	-	-	-	x	-

- permitted  
x permitted, but an abnormal exit  
no not permitted  
[...] see below

- [1] If the procedure is a routine then only "return" may be used; if it is a function then only "resultis" may be used.
- [2] A "signal" in a procedure body may be to only a formal parameter.
- [3] Falling off the bottom of a function body is an error.
- [4] Falling off the "leave" part of a Zahn statement is an error unless there is a "default" label.
- [5] A "fail" is always an abnormal exit, but the "fail finishing" option permits the programmer to specify that certain aspects of a normal exit are to be performed.

Table 3-1: Language Restrictions



to code brackets.

Three principles have governed the decisions documented in this table.

1. It is not acceptable to transfer to a place if doing so makes unclear subsequent flow of control.
2. It is not acceptable to skip over a declaration, thereby jumping into its scope from outside.
3. No transfer is acceptable if it violates stack discipline.

The third point requires further discussion. It is assumed that the COL is implemented using a stack to hold all local (dynamic) items as well as procedure parameters and return linkages. The compiler at all times knows where the base(23) of the current stack frame is, and this base is changed only on invoking a procedure or returning from one. A transfer violates stack discipline if it goes from a place where the stack base has one value to a place where it has another.

Note that it is permitted to transfer into a switch body or into a conditional, since principle 1 stated above is not violated.

---

(23) This is a simplification of the actual case but is adequately accurate to make the point without being misleading.

The table makes it clear that the compiler imposes no restrictions on what the programmer may do within code brackets. This is consistent with the COL's design philosophy that essentially anything that makes sense can be done without code brackets, the latter being reserved for cases in which the programmer knows precisely what he wants. Clearly, a programmer who uses code brackets to "goto" outside of a procedure had best understand the implementation well enough to adjust the stack pointer properly.

## Chapter 4: Expressions

An expression is a constituent of the language that is evaluated for its value. In general, there is no side effect as a result of this evaluation, although there are certain exceptions (discussed below).

Expressions "E" and "NL" and "NC" have identical syntax, but the latter two are used in contexts where a value is required before execution of the program. Expressions "NC" must be evaluable at compile time, and expressions "NL" must be evaluable by the completion of linking. (See section 1.3.3 regarding evaluation of expressions at link time.) It is a reported error if these expressions cannot be evaluated by the required time.

```
208 E ::=                                // expression
209     <primary expression>
210     <operator expression>
211     <aggregate expression>
212     <other expression>

213 NL ::=                               // eval by link time
214     E

215 NC ::=                               // eval at compile time
216     E

--- <primary expression> ::=
    (see section 4.1)

--- <operator expression> ::=
    (see section 4.2)
```

--- <aggregate expression> ::=  
    (see section 4.3)

--- <other expression> ::=  
    (see section 4.4)

The next four sections present in turn the four types of E, and a final section addresses issues of evaluation order.

#### 4.1: Primary Expressions

The primary expressions are the basic constituent parts out of which the programmer builds his programs.

217	<primary expression> ::=	
218	ID	// identifier
219	<integer>	// decimal, octal or hex
220	<floating number>	// floating point
221	<character constant>	// a single character
222	<character string>	// array of characters
223	<logical constant>	// bits
224	true	// boolean
225	false	// boolean
226	nil	// pointer to nothing
227	locked	// a set interlock
228	unlocked	// an unset interlock

--- ID ::=

    (see section 5.1.1)

--- <integer> ::=

    (see section 5.1.2)

--- <floating number> ::=

    (see section 5.1.2)

--- <character constant> ::=

    (see section 5.1.3)

--- <character string> ::=

    (see section 5.1.3)

--- <logical constant> ::=  
    (see section 5.1.2)

An ID is an identifier, a name usable by the programmer to denote some value used in the computation. The syntax of IDs is presented in section 5.1.1.

An <integer> may be expressed in decimal, octal or hexadecimal; a <floating number> may be expressed only in decimal.

The alphabet from which <character constant>s are formed is discussed in section 5.1.3. A <character string> is syntactically a string of characters enclosed in double quote marks. Such a character string is an abbreviation for an array of characters with lower limit one. The special conventions available to quote the double-quote mark, as well as such characters as newline, backspace, tab, etc., are described in section 5.1.3.

The boolean constants "true" and "false" denote the possible values of boolean expressions. Note that there is no commitment in the COL as to how they are represented in the computer.

The construct "nil" is a pointer to nothing at all. The type checking of the language permits it in any context where a pointer is required, but it is an error at run time to attempt to access the item to which it points(24). The value of a pointer may be

---

(24) The compiler does not compile special code to detect this error. However, if practical in the particular implementation, it selects a representation of "nil" such that an attempt to address



compared with the value "nil", using the operators "eq" or "ne". Its purpose is to permit recursive data structures such as lists or trees, since without it and with full type checking there would be no way to get started. The example in section 7.4 shows this.

The constants "locked" and "unlocked" are (the only) appropriate values that may be stored into an interlock. In some cases they may not have well-defined values. For example, some implementors may find it convenient for "unlocked" to be any non-zero value. Interlocks are described in section 3.4.3.

#### 4.2: Operator Expressions

Operator expressions include the infix and prefix operators that are in the language. Rather than present an unambiguous syntax that specifies the relative precedence of all of these operators (a syntax which is both hard to read and quite lengthy), we present instead a simple ambiguous syntax which we then disambiguate with a precedence table.

```
229 <operator expression> ::=
230     E <infix op> E           // infix operator
231     <prefix op> E           // prefix operator

--- <infix op> ::=
    (text below)

--- <prefix op> ::=
```

---

through it causes a hardware error. The "%check pointer" compiler directive causes open code to be compiled for each pointer indirection to detect this error.

(text below)

The classes <infix op> and <prefix op> are exhibited in Table 4-1.

- - - - -

#### Infix Operators

arithmetic:	+ - * / mod **
shift:	lshift rshift lrotate rrotate
relational:	lt < le <= =< gt > ge >= => eq = ne <>
logical:	and or eqv nor xor
boolean:	and or eqv nor xor

#### Prefix Operators

arithmetic:	+ -
logical:	not
boolean:	not

Table 4-1: Infix and Prefix Operators

- - - - -

The <infix op>s may also be used in the special assignment statement syntax on line 148, page 92. The operands of all of these operators are selected from among the basic types. (The basic types are those described in section 2.4.2.) Most of these operators are polymorphic, accepting more than one type of operand.

The infix operators operate on conformable arrays, although not on structures; the prefix operators operate on arrays. Two arrays are conformable if they have the same limits. (See Appendix III for a more precise definition.) Thus "A + B" is acceptable if A and B are conformable arrays whose elements are acceptable operands of "+". Similarly, "not C" is acceptable for any array C whose elements are acceptable operands of "not".

The programmer is cautioned to make no assumptions about evaluation order. The sophisticated optimizing compiler required for the COL feels free to evaluate common subexpressions only once, to alter the order of evaluation of items, to make use of such mathematical laws as commutativity, etc. Read carefully the detailed discussion of this topic in section 4.5.

#### 4.2.1: Precedence of the Operators

The relative precedence of the COL's operators is specified by the table shown in Table 4-2. Phrases in this figure in square brackets, such as "[function call]" near the top of the figure, are keyed to the operators listed in Table 4-1. The term "array selector" on the first line of the table refers to an array access using square brackets. Note on the same line the dot ".", which is the structure selector. Operators on the same line are equally binding and are left associative, except that the relational operators do not associate at all. (That is, "A < B < C" is not defined in the COL.) The last line of the figure affects only

```

MOST BINDING
[function call] [array selector] . @ convert force
**
unary + unary -
* / mod
+ -
[shift operators]
[relational operators]
not
and
or xor nor
eqv
when ... then ... else ...
LEAST BINDING

```

Table 4-2: Relative Precedence of COL's Operators

-----

"else". In the examples of COL text shown in Figure 4-1, all of the parentheses(25) are redundant and could be removed without changing the parsing. Note that the BNF syntax as presented is ambiguous but that the precedence table completely disambiguates any COL expression.

#### 4.2.2: Semantics of the Operators

The table in Table 4-3 shows, for each operator, the permitted operand types as well as the type of the result. Here "num" refers to either integer or floating, with the understanding

---

(25) The COL uses parentheses in expressions in the same way as do essentially all high-order computer languages. The relevant syntax is presented in section 4.4.

(a * b) + c	precedence
(a / b) * c	left associative
-(a**2)	precedence
(x lshift 3) rshift 5	left associative
(a < b) and c	precedence
(A[m])[n]	left associative
(A@).B	left associative
(A[k]).B	left associative
(SP[C])()	left associative
(not a) and b	
(a and b) or c	
when P then X else (Y+Z)	precedence

Figure 4-1: Examples of COL Precedence

All parentheses are unnecessary.

---

OPERATOR	LEFT	RIGHT	RESULT
+ - * /	num	num	num
mod	int	int	int
**	num	int	num
relational	any	any	bool
boolean	bool	bool	bool
logical	log	log	log
shift	log	int	log

Table 4-3: Permitted Operands for Operators

---

that all instances of "num" on any line of the table are to stand for the same type. For relational operators "any" refers to any COL type at all, again with the requirement that both have the



same type.

The following subsections present the semantics of all the COL's operators, as well as of the assignment statement and parameter passing. The latter are not really operators, but they are described here in order to collect all related material in a single place.

#### 4.2.2.1: Semantics of Assignment

The effect of an assignment statement is that the left side is evaluated in L-mode and the right side in R-mode (in whichever order the compiler finds convenient), and then the R-value of the right side is stored into the place pointed to by the left side. If the size and type of the right side agree exactly with that of the left side, there is no problem; the remainder of this discussion concerns possible lack of agreement.

It is a detected error at compile time if the types differ. Some languages provide an automatic conversion of the value on the right to the type on the left; such a conversion is called a coercion. The COL performs no such coercions at run time, since we felt such a situation to be a source of possible error that should be brought to the programmer's attention. However, constants are coerced at compile time to agree with the type of the context. (Details of coercions are in section 6.1.5.)

If the size of the object on the right is smaller than that on the left, the calculated value is stored at the right end of the available space. Logical fields are left-padded with zeros, numeric fields are sign extended, and floating point fields always contain a floating point number in appropriate format.

If the size of the object on the right exceeds the size on the left, the normal effect is to truncate the value by discarding excess bits on the left end. Note that it is the MOST SIGNIFICANT BITS that are lost. There are two ways available to the programmer to deal with this potential loss of information. If the "%warn assign\_size" compiler option is used, the compiler will emit a warning message for each such assignment statement. If the "%check assign\_size" directive is used, the compiler will compile open code to detect a value out of range for the available field. Note that these checks are not equivalent, since the value of the right side may require fewer bits than the size indicates. Given the declaration

```
declare (P8: 8 bit integer, P5: 5 bit integer)
```

the assignment statement "P8 := P5" causes the bits of P5 to be stored at the right end of P8, sign extended. The effect of "P5 := P8" depends on options selected. In the absence of any, the right-most 5 bits of P8 are stored into P5 with possible loss of significant bits. The "%warn assign\_size" compiler directive elicits a warning message from this assignment, and

"%check assign\_size" causes the compiler to compile open code to report as an error a value out of range. Note that it is reasonable to have both of these options in effect simultaneously.

#### 4.2.2.2: Semantics of Parameter Passing

The COL provides three methods for passing parameters to procedures: by read only, by value, and by reference. In a call by read only, the value of the actual parameter is made available to the procedure but cannot be changed. The compiler reports as an error any attempt to change the formal parameter. In particular, it reports as an error an appearance of the formal parameter on the left side of an assignment statement or as actual parameter called by reference. The actual parameter need not be addressable.

Call by value makes a copy of the actual parameter available to the procedure. Effectively, the formal parameter is a local variable of the procedure which is initialized as part of the invocation. Like any other local variable it may be updated within the procedure. Of course, doing so has no effect on the actual parameter. The actual parameter need not be addressable.

Call by reference requires that the actual parameter be addressable. The effect is as if the formal parameter has the same L-value as does the actual. Updating the formal parameter causes the actual parameter to be updated.

In the simplest case, the type and size of the formal and actual parameters are in exact agreement. It is an error detected at compile time if they differ in type. However, the effect of a difference in size is dependent on the call type. For call by read only or by value, a too-small actual parameter is widened as for the corresponding situation in assignment. Similarly, an actual parameter that is wider than the formal parameter is truncated, with the "%warn param\_size" and "%check param\_size" compiler directives having an effect similar to that of the similarly-named directives for assignment. There must be exact agreement of both type and size for an argument called by reference.

#### 4.2.2.3: The Arithmetic Operators

The COL has six arithmetic operators. The operators "+", "-", "\*", and "/" denote addition, subtraction, multiplication and division, respectively. Each is polymorphic, either operating on two integers and yielding an integer result or operating on two floating point numbers and yielding a floating point result. The operator "mod" operates on two integers yielding an integer result which is the remainder of dividing the left operand by the right operand. The operator "\*\*" is a polymorphic exponentiation operator whose right operand (the exponent) must be an integer expression whose value is known at compile time. The result has the same type as the left operand, either integer or floating point.



Each operation on integers takes place in a register at least as wide as the wider operand. The apparent size of the result is the size of the wider operand, but no overflow condition exists unless the result is too wide for the register. Values out of range can be detected only when an assignment is done or a value is passed to a procedure, as described in the previous two sections.

Floating point arithmetic is performed using the representation available on the object computer. If single- and double-precision floating point arithmetic are supported, single is used unless at least one operand requires double-precision.

Both integer division and "mod" yield a positive result if both operands are positive. The effect if either operand is negative is not defined in this document. However, the quotient and remainder always satisfy the standard identity. That is, assuming that all variables are integers, executing

$$Q := M/N; R := M \text{ mod } N$$

calculates values such that

$$(Q*N + R = M) \text{ and } (\text{abs}(R) \leq \text{abs}(N))$$

is true regardless of the signs of "M" and "N".

#### 4.2.2.4: The Relational Operators

For the programmer's convenience, the COL provides several representations for each of the relational operators. These are shown in Table 4-4, each line of the table showing the



lt	<	less than
le	<= =<	less than or equal to
gt	>	greater than
ge	>= =>	greater than or equal to
eq	=	equal to
ne	<>	not equal to

Table 4-4: COL's Relational Operators

-----

representations for one operator. All of these operators are polymorphic, operating on any COL type at all, but both operands must be of the same type.

The equality operators "equal to" and "not equal to" perform just as expected, subject to the following points:

- . It is not useful to ask if two interlocks are equal, since the compiler has freedom to select any values it chooses corresponding to "locked" and "unlocked". For example, any non-zero value may denote "unlocked", so two unlocked interlocks may have different values.
- . If two aggregates are tested, only the defined fields are checked. That is, padding fields do not take part in the comparison.

The conventions defined in section 4.2.2.3 for subtraction are used in comparing two fields of different size. However, an

overflow condition never results from a comparison.

The four relations other than equality and inequality depend on the ordering defined for the underlying data type. These are as follows:

- . integer -- the obvious numeric ordering.
- . floating point -- the obvious numeric ordering.
- . logical -- the ordering is not defined(26).
- . character. The collating sequence of the character set defines the order.
- . boolean -- the ordering is not defined.
- . interlock -- the ordering is not defined. See the comment above about interlocks.
- . condition -- the ordering is not defined.
- . aggregate. The fields are compared from left to right, the first non-equal fields defining the order as described in this discussion. Padding fields do not take part in the comparison.

---

(26) An obvious possibility is to treat a logical quantity as an unsigned integer. However, this is difficult to implement in some hardware architectures for register-sized quantities if the compare instructions insist on treating the left-most bit as a sign.

- . pointer -- the ordering is not defined.
- . function and routine -- the ordering is not defined.
- . general -- the ordering is not defined. Comparing two objects of type general may yield a result different from that obtained on comparing the same two objects when the type is known.
- . subrange. The ordering of the underlying set defines the order of the elements of the subrange.
- . enumeration. The order of listing the items in the declaration of the enumeration defines the ordering. (An item listed before another is "less than" the other.)

The phrase "not defined" in the above listing means that no definition is provided by the definition of the COL. Any implementation defines an order for such items, but it is not necessarily the case that the same order applies in all places or even in subsequent executions of the same program. In spite of this restriction, it is felt that it is still useful to permit comparisons of all COL objects(27).

---

(27) Knuth argues eloquently and convincingly for this view, describing a problem that can be solved in time proportional to the number  $N$  of items if any ordering at all is defined on pointers but requiring time proportional to  $N^2$  otherwise. He has two vectors of pointers and wishes to determine whether or not there are any pointers which appear in both vectors. A simple algorithm is available if the vectors are ordered, but  $N^2$  comparisons are needed if they are not. See point 4 in the letter

## 4.2.2.5: The Boolean Operators

The COL's boolean operators are listed in Table 4-5. The operands and results are all boolean.

---

			and	or	eqv	nor	xor			not
F	F		F	F	T	T	F	T		F
F	T		F	T	F	F	T	F		T
T	F		F	T	F	F	T			
T	T		T	T	T	F	F			

Table 4-5: The boolean (and logical) Operators

## 4.2.2.6: The Logical Operators

The same operators are used for both boolean and logical operations. A logical operator takes two logical operands. If they are not of the same size, the smaller is zero-padded on the left. The result is a bit-wise operation on the two operands.

---

to Hoare in [Knuth-73]. Knuth concludes his discussion with the following remark: "Here is a case where the ordering ... has no semantic meaning, yet my program would work meaningfully (and much faster) if I allowed the machine to order the reference variables in any arbitrary but consistent way." This argument convinced us to define the COL such that all data types are ordered, even though the ordering is not a priori defined.

Table 4-5 defines the operations, with "T" and "F" of that table replaced by "1" and "0", respectively.

#### 4.2.2.7: The Shift Operators

The COL has four shift operators, left and right shift and left and right rotate. For each the left operand is type logical and the right operand an integer. For the rotate operators the left operand must be the size of a register. The nominal size of all results is that of the left operand, although the operation takes place in a register. Zeros are shifted in and bits shifted out are lost.

#### 4.3: Aggregate Expressions

An aggregate is a collection of objects. An ARRAY is an aggregate in which element selection is by an integer subscript; of necessity, each component is of the same type. A STRUCTURE is an aggregate in which element selection is by name; elements need not have the same type.

```

232 <aggregate expression> ::=
233     E [ E , ... ]           // array reference
234     E . ID                 // structure reference
235     <aggregate init>       // value for aggregate
236 <aggregate init> ::=       // value for an aggregate
237     construct ( <aggregate value> )
238     allocate ( <aggregate value> )
239     table ( <type> , E , ... )
240 <aggregate value> ::=
241     <type> , <field value> , ...
242     <type> : E
243     <type>

```



```

244 <field value> ::=                // field of an aggregate
245     <field label list> : E
246     <field label list> ( <field value> , ... )

247 <field label list> ::=            // list of field labels
248     <field label> . ...
249     <subscript list> . <field label> . ...
250     <subscript list>

251 <field label> ::=                // label of a field
252     ID
253     ID <subscript list>

254 <subscript list> ::=
255     <subscript item> <empty> ...

256 <subscript item> ::=
257     [ <subscript> , ... ]

258 <subscript> ::=
259     NC
260     NC .. NC

--- <type> ::=
    (see section 2.4)

```

Line 233 provides the syntax for accessing an element of an array. The multiple subscripting provided by this syntax is a syntactic sugaring for accessing an element of an array of arrays, just as the declaration syntax of section 2.4.3 provides a syntactic sugaring for declaring such an object. An expression such as

A[I, J]

is syntactic sugaring for

A[I] [J]

in which it is the Ith element of A that is subscripted with J. (The parsing rules given in section 4.2.1 make it clear that this parses as

(A[I]) [J]

since subscription is left-associative.) Such an expression makes sense only if each element of A is itself an array. For example, all elements of the array ZAP declared by

```
declare ( ZAP: array[1..10] of array[1..20] of integer)
```

could be set to zero by the code

```
for I in [1..10] do
  for J in [1..20] do
    ZAP[I, J] := 0 // the assignment statement
  endfor
endfor
```

Clearly, replacing the assignment statement by

```
ZAP[I][J] := 0
```

does not change the effect.

Line 234 provides access to components of a structure. The ID in the syntax is the name of a named field, as in the declaration in section 2.4.3. Note that with suitable declarations an access such as

```
A[I].B.C[J]
```

is acceptable. Here A is an array of structures in each of which the B component has a C component which is an array.

Three kinds of expression are provided to initialize an aggregate: the "construct" form, the "allocate" form, and the "table" form. Each has a first "parameter" which is a type, followed by further parameters to initialize fields. For each, those fields named are initialized while other fields are not set at all. The "construct" form builds the object itself, getting

space for it in whatever way the compiler finds appropriate. For example, if the "construct" expression is the entire right side of an assignment statement, the compiled code might well store the named fields directly into the structure specified on the left side. Fields of that structure not mentioned on the right remain unchanged. The "allocate" form makes use of a free storage package, described in section 6.5. For an "allocate" expression the compiler compiles code to invoke a run time free storage function which returns a pointer to a large enough space, and this space is then filled in with the values of the specified fields. It is the user's responsibility to return such space to the free storage package, using the "free" statement described in section 3.4. The "table" expression may be used instead of "construct" for array initialization if the entire array is to be initialized.

Note the doubly-nested "for"-statement on page 149 used to clear the entire ZAP array to zero. The entire example might be replaced by the following single assignment statement:

```
ZAP := construct(array[1..10, 1..20] of integer,  
                  [1..10, 1..20]: 0)
```

Efficient code is compiled for this statement.

The "for" statement given in Figure 3-1 on page 97 might better be coded as

```

X := construct ( array[1..20] of integer,
                [1..5]:0,
                [6]:1, [7]:2, [8]:3, [9]:4, [10]:5,
                [11]:6, [12]:7, [13]:8, [14]:9,
                [15..20]:10
              )

```

but the "table" form is even more convenient:

```

X := table (array[1..20] of integer,
            0, 0, 0, 0, 0,
            1, 2, 3, 4, 5,
            6, 7, 8, 9, 10,
            10, 10, 10, 10, 10
          )

```

Each of these two forms produces the same code, since the right side is evaluated at compile time.

The syntax on line 242 provides for initializing a pointer to a simple type. If for example P is declared to be a "pointer integer", the assignment statement

```
P := allocate(pointer integer: 0)
```

gets from free storage enough space to store an integer, stores zero into that space, and leaves P pointing to it.

Note in lines 248 and 249 that the COL operator "." (dot) is followed by the meta-linguistic symbol "..." to indicate one or more <field label>s separated by dots. Careful reading is required.

Many of the concepts introduced in this section are illustrated in Figure 4-2. This first declares T to be (a type which is) a rather complicated structure and then TA to be (a type which is) an array of T's. T\_ZERO is then declared to be a

```

declare
( // A type for a complicated structure.
  T is structure
    ( P: array[1..4] of
      structure
        ( P1: integer
          P2: boolean
        )
      Q: integer
      R: array[1..7] of integer
    )

  // A type which is an array of the above.
  TA is array[1..2] of T

  // A value to initialize all of T to zero or false.
  T_ZERO = construct
    ( T, P[1..4] (P1: 0, P2: false)
      Q: 0,
      R[1..7]: 0
    )

  // An initial value for a TA.
  TA_ZERO = construct(TA, [1..2]: T_ZERO)

  // Variables of type T and TA.
  V: T
  VA: TA
)

// Some assignment statements.
V := T_ZERO
VA := TA_ZERO

VA[2].P[3].P2 := true
VA[1].P[1].P1 := 7
V.R := construct(array[1..7] of integer, [1..7]: 3)
V.R[3] := 7
V := VA[2]

```

Figure 4-2: Example of Aggregate Usage



constant with a type appropriate to store into an object of type T. In this case it sets all fields of such an object to either zero or false. TA\_ZERO is declared to be a constant suitable to assign to an object of type TA, using the already-defined T\_ZERO. Finally, variables V and VA of types T and TA are declared. The remaining part of the example shows some assignment statements that use all of this. Note that in all assignments the type of the object on the right matches the type of the place specified on the left.

#### 4.4: Other Expressions

The remaining kinds of expression permitted in the COL are described in this section.

```

261 <other expression> ::=
262     ( E )                // parentheses
263     E ( E , ... )       // function call
264     E ( )               // ditto, no arguments
265     max ( E , ... )     // maximum

266     min ( E , ... )     // minimum
267     succ ( E )          // successor
268     pred ( E )          // predecessor
269     abs ( E )           // absolute value
270     truncate ( E )      // truncate down

271     round ( E )         // round-off
272     floor ( E )         // largest int less than
273     ceiling ( E )       // smallest greater than
274     low ( E )           // lower limit of array
275     high ( E )          // upper limit of array

276     convert ( <type> : E ) // convert E to <type>
277     force ( <type> : E )  // assume <type> for E
278     when E then E else E // conditional expression
279     E @                 // contents of pointer
280     explicit ( E )       // do not optimize

```

The semantics of a function call in lines 263 and 264 is very similar to that of a routine call, as described in section 3.1, with binding of actual parameters to formal parameters. The difference is that a function returns a value which is then used in the context in which the call appears.

The built-in polymorphic functions "max" and "min" each take one or more arguments, all of which must be of the same type, the type being any simple one for which the relational operators are defined. The numerically largest or smallest such value is returned. Because these functions are built-in, the compiler is able to generate good code for them.

The functions "succ" and "pred" operate only on an element of an enumeration class, performing respectively the successor and predecessor functions. The ordering is the order of appearance in the declaration of the enumeration. Normally there is no check on going out of range with either of these functions. However, the "%check succ\_overflow" and "%check pred\_overflow" compiler directives may be used to direct the compiler to compile open code to detect this condition. See section 5.3.2.

The polymorphic function "abs" returns the absolute value of its argument. It operates on either an <integer> (or a subrange of the integers) or a <floating number> and returns a value of the same type as its argument.

The four functions "truncate", "round", "floor" and "ceiling" each take a floating point argument and return an integer result. Truncate discards the fractional part, truncating always towards zero. Round returns the nearest integer, with the proviso that a fractional part of one-half is always rounded away from zero. Floor returns the largest integer not greater than its argument, and ceiling returns the smallest integer not less than its argument. This semantics is suggested by the examples shown in Table 4-6. For the values shown in the first column, the

---

	truncate	round	floor	ceiling
-2.6	-2	-3	-3	-2
-2.5	-2	-3	-3	-2
-2.4	-2	-2	-3	-2
0.0	0	0	0	0
1.4	1	1	1	2
1.5	1	2	1	2
1.6	1	2	1	2

Table 4-6: The functions truncate, round, floor, ceiling

---

succeeding columns show the values returned.

The pseudo-functions "low" and "high" in lines 274 and 275 return respectively the lower and upper limit of a discrete type.

The argument may be either a type, a variable whose type is <discrete type>, or a variable whose type is array. They are usually evaluable at compile time.

The syntax provided on line 276 invokes a run time conversion of the expression E to the <type> shown. Conversion is possible among only the basic types: integer, float, logical, char and boolean. Details are presented in section 6.1.6.

The "force" feature of line 277 permits the user to circumvent the compiler's type-checking mechanism. The machine representation of E is assumed by the compiler to have the <type> shown, regardless of what the compiler would otherwise think. Note that no conversion is done -- the only effect is to change what the compiler thinks is the type of an expression. This is always a machine-dependent feature, since the semantics depends on the bit pattern chosen by the implementor.

Line 279 provides a mechanism for dealing with pointers. If E is any expression that evaluates to a pointer, then "E @" is the value of the object that E points to. The object pointed to must be addressable, as that term is defined in Appendix III. A postfix operator is used for this purpose because it frequently requires fewer parentheses in structure references. For example, the expression

A@.B.C@.D

makes sense if A is a pointer to a structure B whose C field is

also a pointer. There is no difficulty reading this, whereas parentheses would be required were a prefix operator used. (This idea comes from PASCAL.)

The explicit expression (line 280) limits the optimization that the compiler performs in a given expression. Further details are presented in section 4.5.

#### 4.5: Order of Evaluation

In general, as mentioned earlier, evaluation of an expression does not have any effect other than returning a value. There are two possible exceptions:

- . If a function is invoked as part of the expression, there may appear in its body assignment statements that update global variables. In the extreme case, the expression itself may be within the scope of such variables. (This is considered to be bad coding style.)
- . Evaluation of the expression may set "event" variables, such as overflow.

The programmer is cautioned not to make any assumptions about the order of evaluation of the constituent parts of an expression, since the compiler in its efforts to compile efficient code selects the order it finds best. In particular, primary expressions may be evaluated in any order or not at all.



The demanding needs of communications programming require that the COL compiler generate code of the highest caliber. The intent of the language design is to place as few constraints as possible on the language that might inhibit the actions of the code optimizer. The basic rationale is that the programmer should not need to be aware of the detailed structure of the object machine in order to be able to write programs that compile efficiently. Examples of the kind of knowledge that the programmer should not be concerned with are these:

- . the best order to access the constituent parts of an expression.
- . special hardware instructions that perform commonly used operations.
- . loop improvements by moving code out of loops.
- . elimination of common sub-expressions.

A good compiler can do a better job of taking advantage of such points than can most programmers. However, in order for the compiler to be able to do so efficiently, it needs the freedom to be able to alter the order of evaluation of some expressions and some assignments. The compiler follows the following rules in performing its optimizations:

1. The compiler is free to use the associative, commutative

and distributive laws when they apply mathematically, with the following exceptions:

- . The "explicit" operator guarantees left-to-right evaluation, except for changes in order that cannot impact in any way on the semantics of the program.
  - . The distributive law is never used in floating point calculations. It is used in integer calculations only if doing so is safe. For example, for integer "A" and "B", it is not correct to replace " $A/2 + B/2$ " by " $(A+B)/2$ ".
2. Any expression whose value can be determined at compile time is replaced by that value. Doing so may permit elimination of branches of a conditional or reducing loops to straight line code or nothing.
  3. Any expression whose final value may be determined by partial evaluation at compile time may be replaced by the known value.
  4. The use of a variable may be replaced by use of another variable if it is known to have the same value at that point. This optimization is never performed for volatile variables.
  5. For variables of type integer or enumeration, uses may be

replaced by uses of another variable providing the effect is the same. For example, if in an iteration on "N" the value "10\*N" is computed within the loop, the compiler may replace that expression by some new variable "T" which it initializes properly and increments by 10 whenever "N" is incremented by one.

6. If the variable updated by an assignment is not later used, the assignment statement may be eliminated. In the previous example "N" may no longer be needed.
7. If no assignments to a variable exist, then the variable may be eliminated. Rule 6 may cause this.
8. The compiler is free to reorganize the evaluation of expressions and order of assignments, just so long as the following hold:
  - . The logical sense of the program remains invariant.
  - . No new exceptional condition such as overflow or divide by zero is introduced.
  - . The frequency of exceptional conditions may be reduced. However, any exceptional condition that would have occurred in a non-optimized version of the program still occurs at least once.
  - . Any function invocation that might have produced a

side effect of local significance is still performed, even if the returned value is discarded.

On occasion the programmer may desire to inhibit some or all of these optimizations. For example, in

$$(a - b) + c$$

it may be important that the compiler not do

$$a - (b - c)$$

even though that is mathematically equivalent, since the difference "b - c" might produce an overflow. To achieve this effect, the programmer might write

$$\text{explicit}( (a - b) + c )$$

The expression after explicit is evaluated from left to right as an entity. Note that

$$(\text{explicit}(a-b)) + c$$

works equally well.

One other mechanism is available to the programmer to control optimization. A variable declared with the "volatile" attribute (line 38) or a structure component with that attribute (line 99) must be accessed from memory each time it is referred to. This is important in hardware architectures in which I/O is initiated by writing an appropriate value into a memory location and status of I/O is determined by reading memory. Consider the (machine dependent) declaration

$$\text{declare (TTY: volatile location(16!C000) 16 bit logical)}$$

for a computer in which writing into (sexadecimal) location C000 controls a teletype. Then the sequence

C := 16#0012; C := 16#043A

must clearly be executed exactly as written. Normally the compiler eliminates the first of successive assignments to the same variable (rule 6 above), but the "volatile" attribute suppresses this optimization in this case.



## Chapter 5: Lexical Matters

The lexical scanner is that part of the compiler that scans the input stream, parsing it into lexical items (lexemes) such as identifiers, punctuations, etc. Those aspects of the language for which it is responsible are described in this chapter.

## 5.1: Identifiers and Constants

An identifier is a name used by the programmer to refer to a value. The COL's constants, those lexemes that denote constant values, include numeric and character and logical constants.

## 5.1.1: Identifiers

An ID is any string consisting of upper- and lower-case letters, of digits, and of the character "\_" (underscore), providing that the first character is a letter. Upper- and lower-case letters appearing in identifiers are treated by the compiler as equivalent. Thus FOO and foo and fOo and FoO all denote the same identifier(28). Further, the programmer may not use as a variable a name such as "Declare" or "DECLARE" which is lexicographically indistinguishable from the reserved word

---

(28) The "%warn spell\_id" compiler directive may be used to request warning messages for such multiple spellings of the same identifier. See section 5.3.1.

"declare"(29). The underscore, which may be used as needed to improve readability of names, is part of the spelling of the ID. Thus "Rate\_of\_Pay" is different from "RateofPay", and "A\_" from "A". Any of "declare\_" or "Declare\_" or "de\_clare" is acceptable as an ID, since they are all lexicographically distinguishable from "declare".

The maximum permitted length of an ID is 31 characters, and all 31 are used in distinguishing IDs. That is, two IDs differing only in the 31st character are distinguished by the COL compiler. It is an error to use an ID longer than 31 characters.

#### 5.1.2: Numeric Lexemes

An <integer> is either a decimal integer, or it is a base followed by a sequence of digits. The base is expressed in decimal and is followed by "!". Letters are used as digits for bases greater than 10, with A or a as 10, etc. Thus 8!77 is the octal constant 77 (decimal 63), and 16!F is the hexadecimal constant F (decimal 15). Note that the first character of an <integer> is always a decimal digit and that the value can be readily deduced in a single scan from left to right. An <integer> may not contain embedded spaces, but it may contain underscore characters which are ignored. Thus each of

100000    100\_000    1\_0\_\_0\_\_0\_00\_\_

---

(29) The compiler directive mentioned in the previous footnote also arms warnings for multiple spellings of reserved words.

represents the same value. The initial character must be a digit.

A floating point number must be expressed in decimal, using the following syntax:

```

<floating number> ::=
    <mantissa> <exponent>
    <mantissa>

<mantissa> ::=
    <int>
    <int> . <int>

<exponent> ::=
    <E> + <int>
    <E> - <int>
    <E> <int>

<E> ::=
    e
    E
  
```

Note that a <floating number> may not begin or end with a decimal point. The class <int> left undefined is the set of decimal integers, with imbedded underscores ignored as in <integer>s. Permitted <floating number>s include the following, where all of the numbers on the same line have the same value:

```

1.0  1.0e0  1.0E0  1E0  1E+0  0.1E1  10E-1
0.00567  567E-5  5.67E-3  0.00567E0
1_000_000E0  1_000_000.0  1E6
0.123_456_789  123_456_789E-9  0.123456789
  
```

Not permitted are numbers such as ".1" or "1." or ".25E-3" which start or end with a decimal point, or numbers like "37.E2" in which the mantissa ends with one.

A <logical constant> is a base followed by the character "#" followed by an integer. Each of the following denotes the same quantity:

8#77 10#63 16#3F 2#111\_111

### 5.1.3: Character and String Constants

A <character constant> is a dollar sign followed by a character, and a <character string> is a string of characters enclosed in double quotes. The default character coding for such constants is ASCII. The programmer may override this default, either for the entire program or for a single constant. The former is accomplished using the "%chars default" compiler directive, described in section 5.3.3. For the latter, the constant is preceded by the name of the character set to be used followed by the character "#". Thus \$A or ASCII#\$A is the ASCII character A (octal 101), while EBCDIC#\$A is the EBCDIC code for A (sexadecimal C1). String constants may be treated similarly, with

EBCDIC#"I AM AN EBCDIC STRING."

being an EBCDIC string. Any implementation of the COL may provide whatever character codings are appropriate for that implementation. In addition, the programmer may define additional character codings in his program, using the "%chars" compiler directive described in section 5.3.3.

The COL includes provision for quoting all ASCII (or other) characters in a way that is easy to type and is readable. By

convention, an asterisk and the characters following it are replaced in a character or string constant by a single character. For example, the following correspondences exist:

*C	carrier return
*L	line feed
*B	backspace
*T	tab
*S	space
**	*
**	"

In addition, \*ddd where each d is an octal digit represents character code ddd. The complete set of the COL's conventions is given in Table 5-1. Where upper-case letters are shown in the table, either upper- or lower-case letters may be used. Note that some characters have more than one coding, such as any of

\*^I    \*^i    \*T    \*t    \*011

denoting tab.

There are additional conventions within <character string>s. A <character string> may not extend from one line to the next unless the programmer makes clear that such is his intention. Within a string constant the compiler ignores \*Z and all space that follows it, where "space" is used here in the technical sense defined in section 5.5.1. Consider the assignment statement

```
STRINGVAR := "I am a very, very, very, very, very, *Z
               very, very, very, very, very, very, *Z
               *svery, very long string!"
```

The string is so long that it extends over several lines of input. Since the programmer wants successive lines to be indented properly for ease of reading, he ends each line with \*Z. Then the



Char	Code	*	
NUL	000	*^@	null
	001	*^A	
	002	*^B	
	003	*^C	
	004	*^D	
	005	*^E	
	006	*^F	
BEL	007	*^G	bell (audible warning)
BS	010	*B	backspace
TAB	011	*T	tabulate
LF	012	*L	line feed
VT	013	*^K	vertical tab
FF	014	*P	form feed (new page)
CR	015	*C	carrier return
	016	*^N	
	017	*^O	
	020	*^P	
	021	*^Q	
	022	*^R	
	023	*^S	
	024	*^T	
	025	*^U	
	026	*^V	
	027	*^W	
	030	*^X	
	031	*^Y	
	032	*^Z	
ESC	033	*X	escape
	034	*^\ _	
	035	*^] _	
	036	*^ _	
	037	*^ _	
SP	040	*S	space
"	042	*"	string quote character
*	052	**	asterisk
@	100	@	commercial at
^	136	^	caret
a	141	*@A	lower-case A
...	...	...	...
z	172	*@Z	lower-case Z
DEL	177	*D	"delete" code

Table 5-1: Character Code \* Conventions

new-line transition and any space on the next line are all ignored. Comments could be similarly ignored. For example, the string

```
"ab*Z    // comment..  
        /* comment */ c*Z  
d"
```

is exactly the same as "abcd".

A character string constant is an abbreviation for an array from one to a suitable upper limit of <char>s. Thus "abc" is just an abbreviation for

```
construct(array[1..3], [1]:$a, [2]:$b, [3]:$c)
```

The former is of course much easier to write and to read. The packing discipline to be used for such character strings is dependent on the implementation.

## 5.2: Macros

The syntax for declaring macros is in section 2.6, further discussion appearing in section 6.1.2.3. However, invocation of a macro is a lexical matter, since a macro may be invoked anywhere in the text of a COL program other than within a string, within a comment, or within a compiler directive. Also, when a macro is declared, the <macro body> is not then scanned: It is merely saved, to be scanned when the macro is invoked.

```
281 <macro invocation> ::=  
282     ID  
283     ID ( <macro arg> , ... )
```

```
284 <macro arg> ::=                // actual param to macro
285   <any character not comma or right paren> <empty> ...

--- <macro body> ::=
    (defined in text below)
```

Note the syntax for <macro arg>, which consists of any string of characters other than comma or right parenthesis. For example, if FOO is a macro, the line

```
FOO(A, B, X=)
```

results in substitution into the body of FOO of "A" for the first parameter, " B" for the second parameter, and " X=" for third.

When a macro name is encountered by the lexical scanner, its definition is examined to determine whether it was declared with parameters. If not, the name is merely replaced by its definition, which the lexical scanner then scans as usual.

If the macro was declared with parameters, its invocation must be as on line 283. In scanning the definition of the macro, each formal parameter is replaced by the corresponding actual parameter. It is a detected error if the number of parameters in the invocation is not the same as the number in the definition.

A <macro body> is a special quoted string. The first non-blank character after the "=" is taken as a quote character and the body is all text up to the next occurrence of that character. Since the programmer may choose his own quote character, there is no special provision for quoting it in the

body.

### 5.3: Compiler Directives

A compiler directive may appear anywhere in the input stream other than within a lexeme or a comment; its effect is to control the compiler in some way. All compiler directives start with the character "%"; most end with the first following semicolon, but a few extend to the end of the line. They are listed in the remainder of this section, some being listed in two places. The compiler directives are not reserved words.

Many of the details of the arguments to the compiler directives are left unspecified in this document. This part of the COL design should be completed by the first implementor.

#### 5.3.1: Compiler Control

The following directives control the operation of the compiler:

% compile fast ; // compilation speed  
This directive specifies that speed of compilation is more important than other criteria and is to be achieved at the expense of run-time speed and/or space, although not at the expense of good diagnostics or correct compilation.

% optimize <arg> ; // specify optimization  
Here <arg> may be either "time" or "space". The compiler is directed to optimize that aspect of the running program at the expense of the other.

% include <file name> ; // include a file  
This directive is replaced by the contents of the named file. The syntax of the file name is dependent on the implementation.

```
% warn <arg> , ... ; // control warnings
    The programmer can activate certain optional warning
    messages. One or more of the following arguments may
    appear, each of which turns on a warning message for the
    condition indicated. The optional argument "off" turns off
    all warnings specified to its right on the line.
    assign_size
        The value on the right side of an assignment statement
        has a size larger than that of the variable on the
        left.
    param_size
        An actual parameter has a size greater than that of
        the corresponding formal parameter.
    spell_id
        An identifier or reserved word is spelled in more than
        one way, such as "FoO" and "fOo", or "declare" and
        "DECLARE".

% list <args> ; // set listing control
    The user is provided detailed control of the listing file
    created by the compiler. (See section 1.3.2.) The details
    for specifying this control are not yet designed.

% linker <args> ; // control the linker
    The programmer is provided control of certain of the actions
    taken at link time. Examples are control of overlay,
    setting the instruction counter, clearing memory, etc.
    Details are yet to be specified.
```

### 5.3.2: Program Development

The user of the COL is provided various aids to help him in program development.

```
% compile fast ; // compilation speed
    This directive, which was described in the previous section,
    specifies fast compilation at the expense of run time
    efficiency.

% assertions <arg> ; // control the use of assertions
    The user may include in his programs assertions --
    expressions which he claims to be true. (See section 6.4.)
    The argument <arg> may be chosen as follows:
    ignore
        Assertions are to be ignored.
    check
        Open code is to be compiled to check assertions, with
```



a run time error indicated if the assertion is not correct.

use The compiler may assume that the assertion is true if doing so will aid it in any way. (For example, it may improve optimization.) This option is implied by "check".

This directive may be used repeatedly in a program to establish different assertion control at different parts of the program. If an assertion appears in a capsule or in an open subroutine, its effect is determined by the assertion control in effect at the place at which the assertion appears, and not at the place at which the code is used. The default in the absence of this directive is that assertions are ignored.

% check <arg> , ... ; // check certain semantic errors

The language definition describes certain semantic errors that are not in general checkable at compile time. This directive may be used to cause code to check them at run time to be compiled. Doing so requires that some error-reporting mechanism be included in the run time environment, as suggested in section 1.3.4. Possible values of <arg> are now listed. The argument "off" causes all checks to its right on the line to be turned off.

subrange

An out-of-range storage into a subrange variable occurs.

subscript

A subscript is out of range.

pointer

A pointer with value "nil" is indirected through.

succ overflow

The built-in function "succ" returns a value out of range.

pred overflow

The built in function "pred" returns a value out of range.

assign size

A value is calculated in an assignment statement whose size exceeds that of the variable into which the store is to be done.

param size

The value of an actual parameter has a size that exceeds that of the corresponding formal parameter.

Other values may also be specified.

### 5.3.3: Language Feature Control

Certain aspects of the language may be controlled by the programmer. Character sets other than those built into the implementation may be specified, and the programmer may specify certain stack parameters.

```
% chars define <arg> ;    // character set definition
    Although the normal character set for the COL is ASCII,
    other sets may also be needed in some applications. Some
    (such as EBCDIC) may be built in, but the user can specify
    others if he chooses. To do so, he gives the name of the
    character set and its collating sequence. The details of
    the <arg> are not specified here.

% chars default <charset> ; // set default character set
    The character set named is used as the default for character
    and string constants.

% stack <arg> ;    // specify stack parameters
    Here <arg> may specify stack location or length. The
    details are not here specified.
```

### 5.3.4: Miscellaneous Features

There are some miscellaneous compiler directives, described below.

```
% message <destination> <text>    // message to user
    The message <text> is sent to <destination>, which may be
    either "terminal" or "error.file". The "<text>" extends to
    the end of the line.

% maintain <args> ;    // debug the compiler
    A collection of compiler directives useful for debugging the
    compiler is to remain in all production versions of the
    compiler. Details are not here specified.

% literal <text>    // suppress macro expansion
    The <text> is used as input to the compiler without being
    checked for the presence of macro names. This makes it
    possible to redeclare or undeclare a macro. The "<text>"
    extends to the end of the line.
```

## 5.4: Character Codes

Although the COL's design is predicated on the full ASCII character set, provision is made for other codes at compile or at run time. For compile time, the input language has been designed so that users of the 64 character subset of ASCII(30) have access to all of the language's features. In addition, the "%chars" compiler directive permits the programmer to define a new character encoding, which he may specify for character constants and string constants. (The compiler directive is described in section 5.3.3; the syntax for character and string constants is in section 5.1.3.) All the ASCII graphical characters are shown in Table 5-2. The column marked "Code" gives the ASCII code (in octal). The column headed "64" contains an "x" for each character in the 64 character subset of ASCII. The characters used in the COL which are missing from the 64 character set are not required to compose COL programs. Lower-case letters are not needed, the lexical scanner treating upper- and lower-case letters as equivalent; and the braces "{" and "}" may be replaced by "begin" and "end", respectively.

---

(30) The 64 character subset consists of those codes from 040 to 137 (octal).

Char	Code	64	Usage in the COL
space	040	x	space
!	041	x	base in an integer
"	042	x	string quote
#	043	x	logical constant; character set
\$	044	x	character constant
%	045	x	compiler directive
&	046	x	[not used]
'	047	x	[not used]
(	050	x	left parenthesis
)	051	x	right parenthesis
*	052	x	operator
+	053	x	operator
,	054	x	comma
-	055	x	operator
.	056	x	struct ref; subrange; decimal point
/	057	x	operator; comment
0-9	060-071	x	digits
:	072	x	label constant; declaration; etc.
;	073	x	statement separator
<	074	x	operator
=	075	x	operator
>	076	x	operator
?	077	x	limit of flexible array
@	100	x	contents of a pointer
A-Z	101-132	x	upper-case letters
[	133	x	subscript delimiter
\	134	x	[not used]
]	135	x	subscript delimiter
^	136	x	[not used]
_	137	x	break in names and numbers
`	140		[not used]
a-z	141-172		lower-case letters
{	173		sequence delimiter
	174		[not used]
}	175		sequence delimiter
~	176		[not used]

Table 5-2: ASCII Character Codes in the COL

### 5.5: Miscellaneous Lexical Matters

This section examines three topics not discussed elsewhere: the treatment of spaces in the input to the COL compiler, the COL's comment conventions, and the rule that permits the user to omit semicolons that occur at the ends of lines.

#### 5.5.1: Spaces in the COL

Any sequence of comments and format effectors (other than within a character string) is treated as a single space. The format effectors are space, tab, carriage return, and line feed, as well as form feed (new page) and vertical tab. In a card oriented implementation the transition from one input line to the next, however it be represented, is taken as a format effector. In general, the programmer need use space only to separate reserved words and identifiers from each other. Spaces are not permitted within reserved words or identifiers, nor inside composite symbols such as "!=" or ".." or "<=". It thus follows that such items may not be continued from one line to the next. Spaces within string constants receive special treatment -- see section 5.1.3.

There are three ways in which the transition from one line to the next is given special treatment:

- . A semicolon at the end of a line may usually be omitted by the programmer, the compiler inserting it automatically.



- . The "//" comment convention is terminated by the end of the line on which it appears.
- . The "%literal" compiler directive acts on the remainder of the input line on which it appears.

In these contexts the end-of-line transition is noted by the compiler. For this purpose, vertical tab and form-feed are treated as end-of-line. Further, a long comment (the "/\*...\*/" form) containing either of these characters is treated as end-of-line.

#### 5.5.2: Comment Conventions

The COL has two comment conventions. For convenient insertion of long comments or very short ones, it uses the PL/I convention that "/\*" introduces a comment which continues (over perhaps many lines) to the first following "\*/". For easy insertion of short comments, "//" introduces a comment that extends to the end of the line on which it appears. Neither comment convention is detected within a character string, except after the special \*Z convention described in section 5.1.3.

#### 5.5.3: Semicolon Insertion

The lexical scanner automatically inserts semicolons between lines as needed(31). For all practical purposes, this means that

---

(31) This feature seems to be very attractive to programmers, particularly those who had previously used assembler. The idea apparently was originated by Martin Richards in his design of

AD-A047 392

BOLT BERANEK AND NEWMAN INC CAMBRIDGE MASS  
COMMUNICATIONS ORIENTED LANGUAGE (COL): LANGUAGE DEFINITION. (U)  
MAY 77 A EVANS, C R MORGAN

F/G 9/2

UNCLASSIFIED

BBN-3534

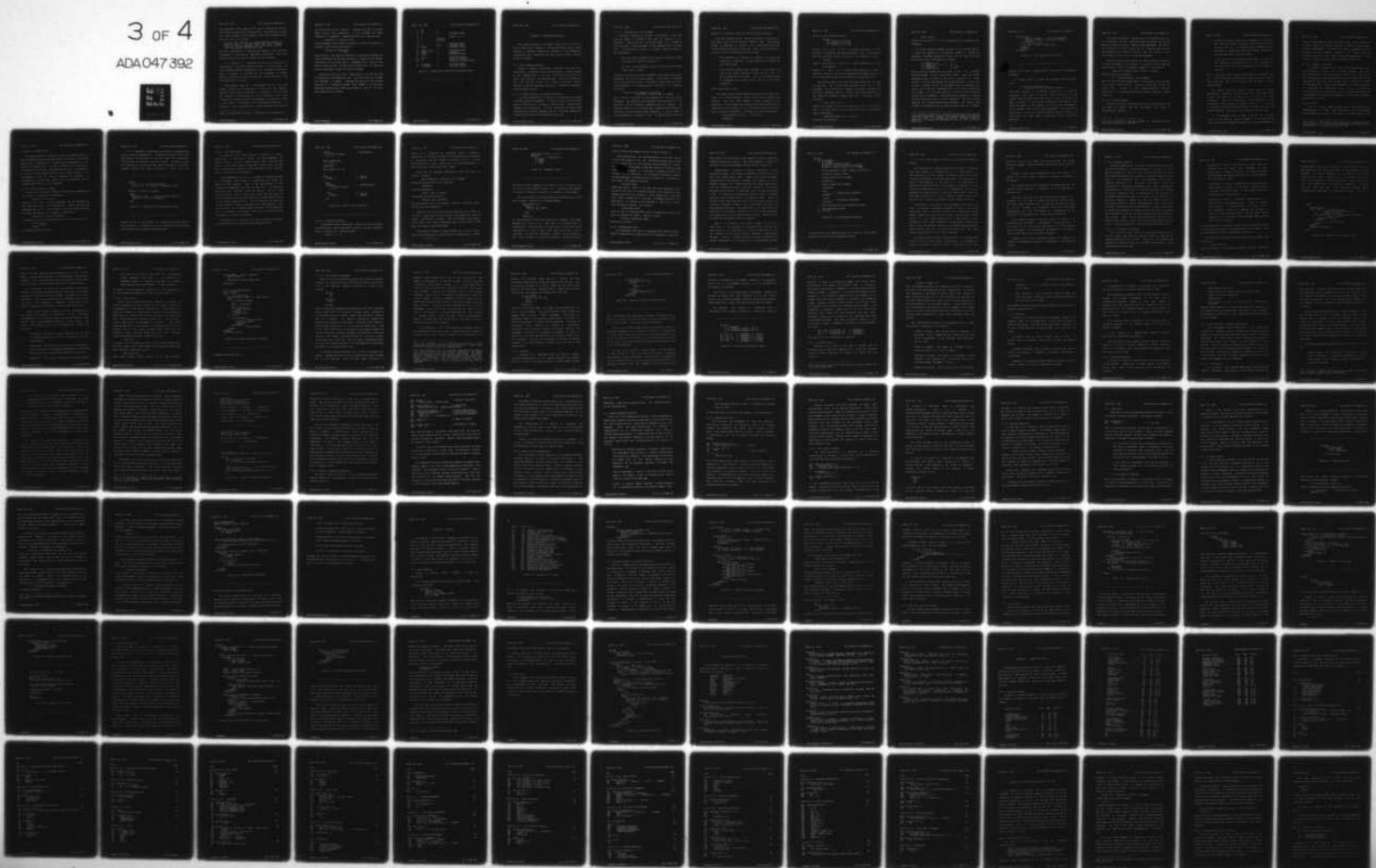
SBIE-AD-E100007

DCA100-76-C-0051

NL

3 OF 4

ADA047 392



most semicolons that appear at the end of lines may be omitted, the compiler automatically inserting them. Thus semicolons hardly ever need to appear in COL programs. The exact statement of the semicolon rule is as follows:

If one line ends with a lexeme that may precede a semicolon, and if the next line starts with a lexeme that may follow a semicolon, then the lexical scanner inserts a semicolon between the lines.

Note that semicolons are inserted only between lines; multiple statements on a line must be separated by semicolons. This rule has been used in most of the examples in this document.

It is important to understand how this rule is applied if a line starts or ends with a macro invocation. In such cases, the rule is applied BEFORE replacement of the invocation by the macro text. This makes it possible for the programmer to deduce semicolon insertion from the written text without having to look at the definition of macros.

Unfortunately there are a few exceptions to the rule just stated, arising from situations in which the rule causes a semicolon to be inserted incorrectly. Fortunately, all such cases are unlikely in the extreme to occur in practice, since they involve COL contexts in which the programmer is unlikely to insert a line break. For example, the rule requires insertion of a semicolon if one line ends with a right parenthesis and the next

---

BCPL; it is also used in EUCLID. (See page 11 of [Lampson-77].)

line starts with an ID. However, a <storage> followed by a <type> might involve this combination, if the <storage> is either "location" or "register". Assuming that FOO is a declared type,

```
declare (X: location(4) FOO)
```

is an acceptable way to declare variable X of type FOO residing in location 4, but writing the above as

```
declare (X: location(4)
        FOO)
```

does not work. The semicolon inserted by the compiler between the right parenthesis and FOO would produce an improper error message. Fortunately it is unlikely that a programmer would insert a line break in such a place, and the other exceptions to the semicolon rule are similarly unlikely to lead to problems.

Table 5-3 lists all of the lexeme pairs in the COL that lead to problems with the semicolon rule. Each line shows two lexemes and (perhaps) a comment. Although the semicolon rule dictates insertion of a semicolon between these lexemes if a line break intervenes between them, there are contexts in the COL for which such insertion is incorrect.

1	ID	(	
2	ID	:	
3	)	ID	<storage> <type>
4	)	(	(when...)(...)
5	)	function	
6	)	routine	
7	)	:	structure syntax
8	)	data	<storage> data
9	label	:	structure syntax
10	<basic type>	:	structure syntax
11	float	(	precision
12	float	:	structure syntax
13	char	:	structure syntax
14	]	(	array of functions
15	]	:	convert; structure syntax
16	<integer>	:	structure syntax
17	<constant>	:	structure syntax

Table 5-3: Lexeme Pairs Violating the Semicolon Rule

-----



## Chapter 6: Miscellaneous Topics

This chapter collects information about topics that do not seem to have a home elsewhere, including language issues, issues relating to separate compilation, inclusion in COL programs of machine-like code, assertions, and the run time library required by the COL compiler.

### 6.1: Other Language Features

Chapter 2, Chapter 3 and Chapter 4 describe in turn the COL's declarations, statement forms, and expressions. Certain aspects of the language that do not fall conveniently into any one of these three chapters are described here. Successive subsections deal with flexible arrays and variadic functions, macros, capsules, exception mechanisms, coercions, conversions, defaults, and environment queries.

#### 6.1.1: Flexible Arrays and Variadic Procedures

A variadic procedure is a function or routine that can accept a variable number of arguments. A flexible array is one such that either or both limit is unknown at compile time. These related concepts are now presented. The next subsection introduces them informally and presents examples, and the following subsections present succinctly the rules governing their use.

## 6.1.1.1: Introduction to the Concepts

A variadic procedure is one that is declared in such a way that the number of arguments that must be supplied is not specified. Because the COL is strongly typed, it is necessary that all except a constant number of the arguments be of the same type. In particular, to declare a variadic function, there are two requirements:

- . The last formal parameter must be marked "variadic", using the syntax on line 29 in section 2.1.
- . The last formal parameter must be a flexible array whose lower limit is fixed.

Consider a routine whose formal parameter list holds three formal parameters, the first two being regular (non-variadic) arguments and the third being variadic, and suppose further that the third formal parameter is a flexible array from one to N of integers, like this:

```
routine PIP(A: integer, B: boolean,  
           variadic C: array[1..?N] of integer)
```

Then in every invocation of the procedure the first two actual parameters must match the corresponding formal parameters in the usual way, and all subsequent parameters must be integers. The effect of the call is as if these last arguments were elements of an array, and the flexible formal parameter N is the number of actual parameters in excess of two. To make this more clear we

explain first flexible arrays and then variadic procedures.

The term "flexible array" refers to an array in which one or both array limits are not known at compile time. The relevant syntax is presented in section 2.4.4, particularly the class <limit> defined starting on line 124. There are two contexts in the COL in which a flexible array may be used:

- . as a parameter to a function or a routine or as the value returned by a function. In such cases the "?" notation of line 126 is used.
- . as an array residing in free storage. In such cases, and only in such cases, evaluation of the limit E in line 125 may be postponed to run time. Such a non-constant limit is permitted only as part of the type in an "allocate" invocation.

We consider these in turn.

The question mark notation may be used only as a limit of an array in a formal parameter list or as the value returned by a function. The ID so used becomes a formal parameter of the procedure, of type integer and called by read only, whose value on entrance to the procedure is the corresponding limit of the array that is the actual parameter. Consider the routine declaration

```
routine FOO(X: array[?L .. ?U] of char)
...
endroutine
```

as well as the array declarations

```
declare
(  A1: array[1..3] of char
  A2: array[0..5] of char
  A3: array[-4..7] of char
)
```

Any one of A1 or A2 or A3 is a suitable parameter to FOO. Whichever is supplied, the formal parameters L and U are bound to the lower and upper limits of the array. For example, the call

```
FOO(A3)
```

causes L to be bound to -4 and U to 7. More interesting, the call

```
FOO("ABCDEF")
```

binds L to 1 and U to 6. (Recall from section 5.1.3 that a character string is an array of characters with lower limit one.) X[1] in this call is bound to the character constant \$A, etc.

Now consider again the routine PIP whose declaration is shown on page 183. The routine thus declared takes two or more arguments, with the first an integer, the second a boolean, and all other arguments integers. Further, the formal parameter N is the number of arguments greater than two. Thus the routine invocation

```
PIP(4, true, -2, 3)
```

causes A to be bound to 4, B to true, N to 2, and C to an array from 1 to 2 such that C[1] is -2 and C[2] is 3, an array which could be denoted by

```
construct(array[1..2], 1:-2, 2:3)
```

Similarly the invocation



PIP(7, false)

causes N to be 0 and C to be an array from one to zero with zero components.

As a more realistic example, consider a function MaxI that is to return the maximum of an arbitrary number of integer(32) arguments. Possible invocations of MaxI are now shown, with the value stored into X indicated by a comment.

```
X := MaxI(-1, 4)           // 4
X := MaxI(1, 2, 3, 4, 5)   // 5
X := MaxI(-62)             // -62
X := MaxI(-5, -8)          // -5
```

Code to define a suitable MaxI is shown in Figure 6-1. Consider the first sample call for MaxI shown above. Here the formal parameter N\_size is bound to the value 1, and N[1] to 4. If the function is called with exactly one actual parameter, N\_size will be zero, the body of the for statement will be executed zero times, and T (the only parameter) will be correctly returned as the answer. It is an error detected at compile time if MaxI is called with no arguments. Since the formal parameter T is marked as being called by value, it can be updated in the body without changing the corresponding actual parameter. Had "value" been omitted, the resulting call by read-only would lead to a compiler

---

(32) As seen in section 4.4, there is actually built into the COL a variadic polymorphic function "max" which accepts any argument type for which an ordering is defined. Because "max" is polymorphic, it cannot be programmed in the COL, but the simpler MaxI can.



```
function MaxI
  ( value T: integer // The first argument.
    variadic N: array[1..?N_size] of integer
  ): integer
  for K := 1 to N_size do
    if N[K] > T do T := N[K] endif
  endfor
  result is T
endfunction
```

Figure 6-1: The Variadic Function MaxI

-----

detected error where T appears on the left side of an assignment statement.

As another example, consider the following formal parameter list:

```
S: array[1..?S_MAX] of char,
  variadic V: array[1..?V_MAX] of general
```

The first argument is a flexible array from 1 to S\_MAX of characters, with S\_MAX an implicit formal parameter of type integer which is set to the actual upper limit of the array as part of the call. V is an array of objects of type general, with V\_MAX the number of such objects. It is up to the programmer to write suitable code to access these parameters. This particular declaration is part of a complete example, given in section 7.2, which shows also some sample invocations of the routine thus declared.

The value returned by a function may be a flexible array, but it is necessary that the compiler be able to compile code at the place of invocation of the function which determines the size of the result(33). For this reason, the COL requires that at least one parameter be a flexible array and that the limits in the result match a limit of such a flexible parameter.

Flexible arrays can be used in one other context, in addition to being a parameter to a procedure. An array in free storage may have flexible limits. For example, suppose P is declared by

```
declare(P: pointer array[1..1000] of integer)
```

Then executing the statement

```
P := allocate(array[1..N] of integer)
```

gets from free storage enough space for the array, with a pointer to it being stored into P. There is a check that N does not exceed 1000. Further, if the "%check subscript" compiler directive is in use, each subscript is checked against N rather than 1000.

#### 6.1.1.2: Flexible Arrays

As mentioned above, a flexible array is one whose limits are not known at compile time. The COL permits such arrays in precisely two contexts:

---

(33) This restriction permits storage for the function to be allocated efficiently on the stack.

- . It may be a formal parameter of a procedure. In that case the limit is declared using the "?" notation and the ID following the "?" is an implied formal parameter of the procedure. A function at least one of whose arguments is flexible may return a flexible result.
- . It may be an array obtained from free space. In that case the limits are supplied as part of the invocation of "allocate", and the space allocated is only as large as required.

Note in particular that a dynamic array cannot be flexible in the COL. The COL has been designed with a stack discipline that is quite efficient, and that discipline does not permit flexible dynamic arrays.

The syntax governing flexible arrays used as formal parameters is from line 126 in section 2.4.4. The question mark in this case must be followed by an ID, not by any other expression. That ID is implicitly a formal parameter to the procedure, of type integer and called by read only. It is bound at invocation to the corresponding limit of the array which is the actual parameter. The flexible array may have any call type.

It is permissible that the same ID be used as a flexible limit of more than one formal parameter. In that case the COL requires that the corresponding limits of the actual parameters be

identical expressions so that the check can be readily performed at compile time. If a function returns a flexible result, there are two requirements: At least one formal parameter must be flexible, and the flexible limits of the result must be identifiers which are also used in flexible parameters(34). Such a flexible result may not be given a static storage class.

#### 6.1.1.3: Variadic Procedures

A variadic procedure is one whose last formal parameter has the "variadic" attribute. Such a formal parameter must be a flexible array with one limit known at compile time. The type of all corresponding actual parameters must have the same type as the components of the array. The flexible limit of the array is bound to the appropriate value, given the number of actual parameters and the known other limit of the array. (In the usual case the lower limit of the flexible array is one so that the upper limit is the number of extra actual parameters.) The variadic parameter may have any call type.

#### 6.1.2: Macros

The COL provides three different macro-like constructions: abbreviations for types, open subroutines, and conventional string-substitution macros. Each serves a different purpose and

---

(34) These two requirements make it possible to determine before the invocation the size of the returned value so that space can be allocated for it on the stack before the call.



is used in a different way.

The terms "free" and "bound" are used in a technical sense in the following discussion, and the reader is advised to study their definitions in Appendix III. Saying that an appearance of X in a macro is free means that X is used in the macro but that there is no declaration in the macro that governs that use. In particular, X is not a formal parameter of the macro. Recall that it is a requirement of the COL that all variables be declared. An alternate way to state that rule is that it is required that no variable appear free in any COL program.

#### 6.1.2.1: Abbreviations for Types

Section 2.2.3 provides a syntax for declaring an abbreviation for a type. For example, the declaration

```
declare ( R is [1..20] )
```

declares R to be the indicated subrange. The COL reserved word "is" indicates such a type abbreviation. Within its scope the programmer may use R in any context in which he might otherwise have used the subrange. For example, he might write

```
declare ( H: array R of integer )
```

to declare H to be an integer array from 1 to 20, and then write

```
for N in R do  
  ... H[N]...  
endfor
```

to iterate through the array.



It is important to understand the binding of variables that occurs in such an abbreviation. Variables occurring free in the abbreviation are evaluated at the point where the declaration occurs and not at the point where it is used. The idea is that the identifier is to have the same meaning throughout its scope. Consider the COL text shown in Figure 6-2. Both T and U are

```
-----  
  
declare  
  ( K = 10  // Declare a constant  
    R is [1..K]  // Declare a subrange (a type)  
  )  
  ...  
declare ( T array R of integer )  
  ...  
  begin  
    declare (K = 8)  // Hole in scope of outer K.  
    declare ( U array R of integer )  
    ...  
  end
```

Figure 6-2: Binding of Free Variables in Types

-----

arrays from 1 to 10, even though U is declared at a point where K has the value 8. In section 6.1.2.3 it is pointed out that string substitution macros produce a different effect.

## 6.1.2.2: Open Subroutines

By using the word "open" before a function or routine declaration, as described in section 2.1, the programmer may specify that it is to be compiled as open code. This buys time saving at run time, usually at the cost of space, since the time cost of subroutine linkage is saved but there may be multiple copies of the body of the procedure.

As was mentioned in section 2.1 on page 44, changing the mode of a procedure from "closed" to "open" does not change the semantics of the program. Some of the implications of this property are suggested by the code in Figure 6-3. (The macros in that figure are discussed in the next section.) TOM is an open routine in which N and X appear free. Were the mode closed, it would be obvious that each call would have the same semantics with N and X bound to declaration #1. The COL requires this effect for an open routine, so call #2 must have this same semantics. Thus the fact that call #2 appears within the scope of declaration #2 is of no relevance. In the next section it is pointed out that the effect for a macro is different.

The effect of an assertion in an open procedure is precisely the effect of that assertion were the procedure closed.

```

declare                                // DECLARATION #1
( X: static integer
  N = 4
)

open routine TOM()
  ... N ...
  ... X ...
endroutine

macro HENRY = "X + N"

...

TOM()                                // CALL #1
... HENRY ...                        // USE #1
...

begin
  declare                                // DECLARATION #2
  ( X: static boolean
    N = 6
  )
  ...
  TOM()                                // CALL #2
  ... HENRY ...                        // USE #2
  ...
end
...

```

Figure 6-3: Scope in Open Procedures

-----

#### 6.1.2.3: Conventional Macros

Conventional string substitution macros are also available, with declaration syntax described in section 2.6 and invocation syntax in section 5.2. The declaration

```
macro R = "1..K"
```

causes R to represent the indicated string. Subsequent appearances of R within the scope of this declaration are replaced by the string. Note the example in section 6.1.2.1. If R had been declared as a macro rather than as a type abbreviation, U would have been an array from one to eight.

There may be parameter substitution when the macro is invoked. Consider

```
macro VECTOR(N) = "array[1..N] of integer"
```

Within the scope of this macro, the text

```
VECTOR(20)
```

would be replaced by

```
array[1..20] of integer
```

A possible use of this macro is

```
declare ( FOO: VECTOR(8))
```

One could easily write a more general macro in which both limits and the type were parameters.

Figure 6-3 in the previous section exhibited the effect of the COL's scope rules on open routines, and pointed out that the effect is different for macros. Use #2 of HENRY in that figure occurs within the scope of declaration #2, so the X and N in the macro are bound to that declaration.

The example in Figure 6-4 shows another way to use a macro. After macro substitution is done on the line defining A, it reads

```
A = N; N = N + 1
```

```

macro NEXT = "N; N = N + 1"
declare
(  N = 1    // Initialize it.
  A = NEXT
  B = NEXT
  C = NEXT
)

```

Figure 6-4: Example of Macro

- - - - -

The effect of this fragment is to define A to be 1, B to be 2, and C to be 3; it also leaves N equal to 4. The scope rules defined in section 2.7 permit a constant to be redefined in this manner.

Macro invocations can produce unexpected bindings that can be hard to debug. Consider the following example:

```

macro POP(A, B) = "
  begin
    declare (N: integer)
    ... N ... A
    ...
  end"
...
POP(N, W)

```

In the invocation (in the last line of the example), the formal parameter A is bound to N. Thus each A in the macro body is replaced by N. But each such A in the body is in the scope of the declaration of N that is part of the macro definition. This is presumably not what the programmer intended and is therefore a bug. Care is needed to avoid this situation, particularly since



there is little the compiler can do to help to detect it.

Macro declarations are block-structured in the same way as are all other names in the COL. The string substitution is done in the lexical part of the compiler, which precedes the parser. The result of this strategy is that ANY instance of the macro name other than a comment or a string or a compiler directive is replaced by its definition. It is therefore necessary to have a special mechanism to enable the user to redeclare or undeclare a macro name. The compiler directive

```
%literal <text>
```

causes the <text>, which is all the text after "%literal" up to the end of the line, to be passed directly to the remainder of the compiler without being scanned for macro substitution. Note the macro used in Figure 6-4 on page 196. Presumably neither N nor NEXT need to be defined in the remainder of the block in which this declaration appears. Just writing

```
undeclare(N, NEXT)
```

would not work to undeclare NEXT, since its appearance on the line would be expanded as a macro. However, writing

```
%literal undeclare(N, NEXT)
```

would work correctly.

### 6.1.3: Encapsulated Types

An encapsulated data type is a mechanism that permits certain types of implementation safety that are not otherwise attainable.

Historically, the idea seems to have appeared in SIMULA, where the "class" provides the facility. The idea reappears in EUCLID as "records", the COL implementation being similar to EUCLID's.

Syntactically, a capsule is merely a structure in which declarations appear. The syntax is presented in section 2.4.3, to which the reader should refer. Figure 6-5 provides an example showing most of the features, although the example is not a complete program. The example consists of a single declaration. First F00 is declared to be a structure, and then V1 and V2 are declared to be variables of that type. Now note the declaration of F00. It starts off conventionally with two fields, an integer labelled A and a boolean B which is initially false. Thus V1.B has the initial value false as soon as V1 is declared, but V1.A is initially undefined. Fields C and D are similar but have the extra modifier "private". This means that C and D are known only internally in the capsule, so that V1.C or V1.D are not permitted, even though both V1 and V2 have these two fields. (We see shortly how they can be accessed within the capsule.)

Next E is declared to be a static integer. It is not a field, since it is declared in a declaration and not with the field syntax of an ID followed by a colon, so that there is only one instance of E no matter how many variables of type F00 are declared. Since E is marked private it cannot be accessed from outside the capsule. F is similar but, not being marked private,

```
declare
( FOO is structure
  ( A: integer
    B: boolean initially false
    private C: interlock initially unlocked
    private D: static array[1..4] of integer

    private declare(E: static integer initially 0)

    declare(F: static integer)

    private function S(): integer
      ...
    endfunction

    public routine T(N: integer)
      ...
    endroutine

    start
      ... // Initializing statements.
    endstart

    finish
      ... // Finalizing statements.
    endfinish
  )
// End declaring FOO as an encapsulated type.
// Some instances of FOO:
V1, V2: FOO
)
```

Figure 6-5: An Encapsulated Data Type

-----

is known outside the capsule as well as inside it. This item is referred to as FOO.F from outside the capsule.

S is a private function and thus is known only inside the capsule.

The whole point to capsules comes from the public procedures that they contain, as exemplified by T in the example. A reference such as V1.T(N) invokes the routine T with V1 as a sort of implicit parameter, in the following sense. Any instance of a field of F00 appearing in the body of T is interpreted as referring to that field of V1. For example, an appearance of A or C would be interpreted as V1.A or V1.C, respectively. This is the mechanism that permits access to private fields, there being no other way to access V1.C or V1.D. Within T there could be an invocation of S, with an inheritance of this naming convention.

The reserved word "start", which may appear only at the top level of a capsule declaration, signifies a piece of code to be obeyed each time a variable of type F00 is declared. In the present example the code would be obeyed twice, once for V1 and once for V2. The naming convention of T applies here, in that instances of field names encountered in obeying the "start" code refer to the corresponding fields of the variable being declared. Fields marked with the "initially" attribute will have been initialized before the "start" code is entered.

The reserved word "finish" introduces code to be obeyed when V1 and V2 are no longer needed. Specifically, it is obeyed on a normal exit from the scope of V1 and V2. Note that a "fail" (see

section 6.1.4.3) is not a normal exit and can permit the finish code to be skipped, unless the "fail finishing" form of line 207 is used.

Although a macro may be declared at the top level of a capsule, it is never public. That is, its scope is confined to the capsule.

The initialization and finalization attributes may not be used for a static variable. An attempt to do so is detected as a compiler error.

Note that there are two ways to cause initialization in a structure. If all that is needed is to set one or more fields, the "<init value>" attribute may be used with the field, as shown on line 99. However, the programmer with more complicated requirements than can conveniently be accommodated by the evaluation of an expression may use the "start" mechanism.

If an assertion appears at the top level in a capsule, it is assumed to be an invariant of the capsule. Such an assertion is to be true at all times except when within one of the capsule's public procedures. In checking mode, the assertion is checked on entry to and return from each public procedure.

Section 7.5 contains a completely worked out example of a capsule for implementing queues in a safe manner. This should be studied.



#### 6.1.4: Exception Handling

Exception handling has to do with processing events that are out of the ordinary in a given application. The idea is to be able to optimize the code so as to handle efficiently the usual case while providing for the unusual in a way which hopefully adds little or no additional cost. The problem to be solved is exception handling in a structured environment. While the discipline of structured programming has made unquestioned contributions to the ability to construct programs whose effects are readily deducible from an examination of the listing, an unpleasant byproduct has been an increase in the difficulty of handling exceptions. Needed are ways to detect and act on exceptional conditions without imposing extra cost (or at least very much additional cost) on the handling of usual conditions. To assist the programmer in solving this problem the COL provides three different tools, each of which attacks different aspects of the problem. These are the event declaration (section 2.3), Zahn's device (section 3.4.2), and the failure mechanism (section 3.4.4). They are discussed in the following three subsections.

##### 6.1.4.1: The "event" Declaration

At a very local code level, it is sometimes necessary to be able to detect and branch on exception conditions detected by hardware events such as overflow. Almost all CPUs provide some mechanism to detect overflow, but the method varies widely. Three mechanisms cover most cases:

- . A status bit is set by any instruction that causes an overflow, the bit being left unchanged by most instructions. To detect overflow in an area of program the programmer need merely clear this bit at the beginning and interrogate it later.
- . A status bit is set or cleared by any instruction that might cause an overflow. It is harder to detect overflow in an area, since the bit must be interrogated after each instruction that might alter it.
- . Overflow may cause an interrupt. Usually there is a status bit the programmer may set to enable or inhibit this interrupt. To detect overflow in an area of code, the programmer enables the interrupt and arranges to "catch" any interrupt that occurs. This situation can be reduced to the first case if the interrupt routine merely sets a flag and returns.

In designing the COL we sought a mechanism that interfaces to each of these kinds of hardware in a reasonably efficient manner. This is the "event" declaration, described in section 2.3.

We continue to use the overflow condition as an example. The declaration

check overflow

does two things: It declares a boolean variable "overflow",

initialized to false, and it specifies that within the scope of the declaration the overflow condition is to be detected by the hardware. Any use of "overflow" as a variable in that scope returns true or false, as there has or has not been an overflow since the declaration or since the programmer set the variable by code. The example shown in Figure 6-6 is suggestive. Although the code looks more like a data processing program than a communications application, it illustrates the point. This is a

```
-----  
  
begin  
  ...  
  check overflow  
  for ... do // Do all the cases.  
    ... // Fetch data for the next case.  
    ... // Process the data.  
    test overflow do // Has there been overflow?  
      ... // Report bad data.  
      overflow := false  
    otherwise  
      ... // Report answers.  
    endtest  
  endfor  
  undeclare overflow  
  ...  
end
```

Figure 6-6: Example of the "overflow" Event

simple loop that fetches data, processes it, and then reports answers. Any overflow occurring in the processing is detected and reported. Note that overflow is set to false after having been found to be true so that the next case will be treated properly.

It is important to note that the overflow condition is detected only in that code that is within the scope of the event declaration, since the usual COL scoping rules apply to an event declaration. Any overflow occurring in a procedure called within that scope is not detected or reported. A convenient way to detect and report such overflows is shown in the next section.

Certain <event> IDs are defined in all COL implementations, and others are defined as appropriate in any particular implementation. Note that the <event> IDs are not reserved words but are merely IDs presumably defined in an outer block. The <event> IDs defined in all COL implementations include the following:

- . conversion. This event is an error detected at run time on converting one type to another. See section 6.1.6 for more details.
- . assign\_size. This event occurs when a value calculated on the right side of an assignment statement has a value that does not fit into the place specified on the left side.
- . param\_size. This event occurs when an actual parameter

has a value too large for the size of the corresponding formal parameter. The check is not performed for parameters called by reference, since the COL requires that there be exact size agreement for such parameters.

The <event> IDs that might be predefined in a COL implementation for the Lockheed SUE computer are shown in section Ap2.1.5.

#### 6.1.4.2: Zahn's Device

Zahn's device, particularly when conditions are passed as actual parameters, provides another way to handle exceptions. This mechanism is described in section 3.4.2. As an example, we expand the example of the previous section to detect overflows that occur in a routine that is called as part of processing the data. The modified code is shown in Figure 6-7. Here we have a routine WORK whose last parameter is a condition. The actual parameter BAD\_DATA is a Zahn signal, corresponding properly to the declared type of the formal parameter OVFL. Note that the "overflow" event is enabled within WORK and that a detected overflow terminates execution of WORK as soon as

signal OVFL

is obeyed, with an immediate transfer to the case label BAD\_DATA. With this coding, such a signal might be used anywhere within WORK where bad data is detected, just as

signal BAD\_DATA

might appear anywhere within the body of the Zahn statement



```
routine WORK(..., OVFL: condition)
  check overflow
  ...
  if overflow do signal OVFL endif
  ...
endroutine

...
// ...the processing loop...
check overflow
until ... do
  ... // Fetch data.
  upon BAD_DATA or DONE // Zahn's device.
  leave
  // Process the data.
  ...
  WORK(..., BAD_DATA)
  ...
  test overflow do
    signal BAD_DATA
  otherwise
    signal DONE
  endtest
do
  case BAD_DATA:
    ... // Report bad data.
    overflow := false
    stopswitch
  case DONE:
    ... // Report answers.
  endupon
enduntil
undeclare overflow
...
```

Figure 6-7: Zahn's Device and "overflow"

-----

(between "leave" and "do").

## 6.1.4.3: The "failure" Mechanism

There is one final mechanism provided for exception handling, designed for disastrous errors that abort the entire application. This is the "failure" feature, described in section 3.4.4. Consider

```
failing
  S1
  S2
  ...
failhere
  SF
  ...
endfail
```

Once the "failing" statement has been passed, any "fail" statement that is obeyed until the "failhere" is passed causes statement SF to be obeyed, while if control reaches that "failhere" normally then SF is skipped. The important distinction between this mechanism and the two just described is that any "fail" obeyed in any procedure called by S1 or S2 will have the same effect. That is, any "fail" that occurs in any part of the program between the time of passing the "failing" statement and passing the "failhere" causes SF to be obeyed. Note the distinction: An event is scoped in the usual way to the COL text that appears immediately after it, while the failure mechanism is time-dependent during program execution in a totally different way.

A relatively simple mechanism can be used to implement this feature. Assume that there are two variables whose scope includes all COL code compiled: FAIL\_L of type label, and FAIL\_S of

whatever type is appropriate to hold a stack pointer(35). The effect of "failing" is to save the present contents of the two variables in local dynamic storage (i.e., on the stack) and to store into FAIL\_L the address of SF and into FAIL\_S the current stack pointer. Passing the "failhere" part of the failure statement causes the two variables to be restored to their old values as saved. The "fail" statement is merely a special sort of "goto" that sets the stack pointer to the value in FAIL\_S and then goes to FAIL\_L. The code for both "failing" and "fail" could be programmed in the COL in code brackets, but the intent is to provide this feature in a machine-independent way. This feature is modeled closely after the BCPL routine "LongJump"(36). However, "failing" does more for the user in one sense, as well as being more restrictive in another.

It is important to note that the failure mechanism causes a non-standard exit from many places. Certain applications expect a standard exit, with resulting cleanup and finalization. For

---

(35) This discussion is not compatible with the COL's strong typing, but it should nonetheless be instructive. It makes some plausible assumptions about the implementation.

(36) The procedure library for most BCPL implementations contains the function "Level" and the routine "LongJump". The former merely returns the current value of the stack pointer. The latter takes two arguments, one a label and the other a value returned by "Level". It transfers to the label with the stack pointer set. These procedures were part of Martin Richards' original implementation of BCPL. They are described on page 92 of [BBN-74].

example, any variable whose type is a capsule may have finalization specified on exiting the scope of the variable. Such finalization is skipped if a "fail" causes the code never to be exited in the normal way. Suppose FOO is a capsule that contains finalization, and consider the routine STUFF like this:

```
routine STUFF(...)
  declare (V1, V2: FOO)
  ...
  THING(...)
  ...
endroutine
```

Any normal return from STUFF is to cause the finalization for V1 and V2 to be performed. Now note that STUFF calls THING, and suppose THING contains a "fail" statement. (The failure is presumably caught in the code that invokes STUFF.) The effect is that the finalization in STUFF is never performed. This may be acceptable, since "failure" is intended for catastrophic situations in which there is no hope for recovery. However, the programmer can solve this problem if he chooses, as shown in Figure 6-8. With the structure shown, any failure in THING (or in any procedure it calls) will be trapped to the "fail finishing" statement after the "failhere", which in turn will lead to the proper place.

#### 6.1.5: Coercions

A coercion is a conversion from one type to another, performed automatically by the compiler because of the context. That is, a value of one type is used in a context where another

```
routine STUFF(...)
  declare (V1, V2: F00)
  failing
    ...
    THING(...)
    ...
  failhere
  fail finishing V1, V2
endfail
endroutine
```

Figure 6-8: Capturing a "fail" for Finalization

-----

type is required and is therefore converted. The philosophy in the COL is to have very few coercions, and those only where doing so is clearly "safe". This point requires elaboration.

There are two contexts where coercion is performed: across an assignment statement, and in procedure invocation. In the former it is performed if the type of the value on the right differs from that of the variable on the left, and in the latter it is required if the type of the actual parameter differs from the declared type of the formal parameter.

The basic rule on coercion is quite simple: NO TYPE COERCION IS DONE IN THE COL. That is, there is no automatic conversion among integer, logical and boolean. However, this rule applies to run time conversion only. The COL compiler converts constants as required at compile time. For example, if X is a floating point



variable, an appearance of an integer constant in an expression such as "X + 1" or a statement such as "X := 1" is converted by the compiler to floating.

Size coercion, on the other hand, is performed. Integers are widened as needed with sign extension to fit the context. Logicals are similarly widened, storing a narrow logical quantity right-justified in a wider field with leading zeros as needed.

The situation with respect to "different" types is illustrated by the example in Figure 6-9. First three types are

- - - - -

```
declare
(  T  is integer;           N:  T
  T1 is different integer;  N1: T1
  T2 is different integer;  N2: T2
)
N  := N  + 1  // statement 1 -- correct
N1 := N1 + 1  // statement 2 -- correct
N1 := N1 + N  // statement 3 -- illegal
N2 := N2 + N1 // statement 4 -- illegal
N2 := N1      // statement 5 -- illegal
```

Figure 6-9: Coercion of "different" Types

declared: T, which is a synonym for integer, and T1 and T2, which are different from integer and from each other. A variable of each type is then declared, Statement 1 is correct, being ordinary integer arithmetic. Statement 2 is correct even though the left operand of + is a different integer and the right operand is an integer, since the integer constant 1 is coerced to be a different integer. However, statement 3 while similar is not correct, since the coercion is not performed on a variable of type integer. The illegality of statement 4 shows that T1 and T2 are not only different from integer but also different from each other. Statement 5 shows that the coercion is not performed even across an assignment statement. The effect apparently intended by each of the three illegal statements can be had by using "convert", like this:

```
N1 := N1 + convert(T1: N)    // statement 3
N2 := N2 + convert(T2: N1)   // statement 4
N2 :=      convert(T2: N1)    // statement 5
```

(See section 4.4 for discussion of "convert".)

#### 6.1.6: Conversion Rules

Conversion of an item of one type to another type is performed only if the user requests it(37), using the "convert" built-in function described in section 4.4. Recall that the syntax is

---

(37) This refers only to run time conversion, since constants are coerced at compile time.

convert ( <type> : E )

The code compiled converts the expression E so as to retain its value but to be represented as for the <type> given. (Of course, this conversion is performed at compile time if the compiler is able to know the value of E.) The conversions are performed at run time either by open code or by a library routine, depending on the characteristics of the object computer. In some cases errors may be detected in the conversion, such as argument out of range. Within the scope of the "check conversion" declaration such errors set the event "conversion" to true, while otherwise they are ignored.

Each admissible conversion is now discussed briefly, along with mention of possible error situations.

- . integer to float. The obvious numeric conversion is performed. If the integer has more significant bits than can be accommodated in the mantissa, least significant bits are lost.
- . float to integer. The number is converted (with a conversion error if it is out of range).
- . character to integer. The result is the encoding of the character in its character set. For example, the value of "convert(integer: ASCII#\$A)" is 8!101.
- . integer to character. This is the reverse of the previous

conversion. It is a conversion error if the integer is out of range.

- . size conversion. This conversion does not change the value but might be used to elicit a conversion event if the value were out of range.

This document does not define the value returned in the event of a detected conversion error.

#### 6.1.7: Defaults

A default is a decision made by the language designer in a situation where the programmer has a choice which includes the option of not saying. The basic design philosophy on defaults in the COL is that there are very few of them. All of them are now listed.

A procedure may be compiled either open or closed. If neither mode is specified, the default is that the compiler chooses.

Procedure parameters may be called either by read only, by value, or by reference. If nothing is specified, the default is by read only.

Elements of an encapsulated data type may be either public or private. Default is public, if neither is specified.

In declarations of variables of type char and in character and string constants, the default character set is ASCII.

The programmer may specify a size in declarations of logical, integer, and floating quantities. If he does not, an implementation-dependent default is used. This default is selected to be most appropriate for the hardware on which the program will run. The expected default for numeric data is a word, and for a logical it is the smallest addressable unit (usually a byte).

If no storage class is specified for a scalar variable, a procedure parameter, or the returned value of a function, the default is dynamic.

In the absence of a "%assertions" compiler directive, assertions are ignored by the compiler.

Note that there is not a default initial value for scalars or fields of a structure declared without the initial attribute. There is no assumption that the programmer can safely make about the value of such an item.

#### 6.1.8: Environment Queries

A running COL program is able to make certain queries about its environment. Indeed, most of these queries can be made at compile time. Some of the items that can be queried are the following:



- . word size and byte size.
- . default size for integer, logical, character, float.
- . number of hardware registers.
- . memory size.
- . the limits (if any) on the block size available from allocate.

Access to these quantities is via certain identifiers declared to be constants that are automatically available to the programmer. The remaining details (such as the names used) are not specified in this document.

## 6.2: Issues Relating to Separate Compilation

In any programming effort whose size is much larger than trivial, the complete program is broken into separate pieces, referred to in this document as MODULES, each of which is compiled separately. The rather complex issues relating to the problem of separate compilation are discussed in the remainder of this section. The next subsection describes the problems to be solved, as derived from the COL's design goals, followed by an overview of the solution; the following subsection presents in detail the relevant syntax and the associated semantics; and finally, some ideas about implementation are presented.

### 6.2.1: Introduction

It is necessary that there be communication between modules compiled separately, communication of both program structure and

data structure. Program structure involves program elements in one module that can be accessed from another. For the most part such elements are functions and routines, although labels can be similarly accessed(38). COL's type checking rules require that all parameters be fully declared, as must be the value returned by a function.

Data structure involves access to external data bases. The type checking rules of the COL require that the types of such external data be declared so that the compiler, aided by the linker, can check them. The data referenced may be in a module along with other code or in a separate module. In either case, the COL provides adequate syntax for describing such data along with their type and, if desired, initial values. In particular, it is not necessary to use an assembler to provide for initialized data bases.

Two principles have governed the design of this part of the COL.

Strong typing. ALL references are checked as to type before execution. In particular, references to data in other modules are so checked, as well as parameters to external procedures.

---

(38) No "goto" statement may lead to such a label unless that "goto" statement appears within code brackets.

Only one copy of any given data description. In a programming effort involving many programmers and many modules, there is a worthwhile payoff from having in existence only one copy of key information.

The first point has been discussed previously; the second deserves further attention. Consider a data base that is referenced by several modules. Each datum in it has a type, and each module using it must know that type. It is good management for there to exist only one copy of the COL description of the data base, since otherwise there is no way to change that description that is both practical and safe. The COL is designed so as to meet this requirement.

An additional problem must be solved. Consider a large project involving many modules and many shared data bases, and assume that a particular data base is to be used by only a few modules. There are two issues: First, it is desirable that the names used in the data base not clutter up the name space of the other modules. Second, making these names (and therefore the data base) inaccessible from other modules makes impossible certain kinds of programming errors, with immediate advantages for more nearly error-free programming and probable long term advantages for program verification. The COL provides a mechanism designed to solve these problems.

These then are the problems to be solved. Before presenting the details of the solution chosen for the COL, it seems useful to present that solution first in overview. Figure 6-10 shows in outline form three modules, BILL, BILL1 and BILL2, which taken together are assumed to provide certain services to the rest of some larger system. Note the first module, named BILL. It starts with a "module" statement giving the name of the module, followed by two "public" statements that specify which items in BILL can be known outside of BILL and to whom. The first specifies that BILLF1, BILLF2 and BILLR1 are known without restriction (i.e., to all modules that choose to "examine" BILL), and the second specifies that FN1 and RT1 are to be known to only BILL1. Next the "examine" statement directs the compiler to look at the named modules to determine what entries they provide that are to be known to BILL. In the present case we see that BILL1 makes FN11 and RT11 known to BILL and that BILL2 makes known several types and some data. Module LIBRARY, not shown here, contains a collection of useful utilities available to all.

The effect of all this is that only BILLF1, BILLF2 and BILLR1 can be accessed from elsewhere; the others can be accessed only within the modules shown(39). Note that module BILL declares three functions and two routines. The first three are listed as

---

(39) It is important to realize that the phrase "can be accessed only..." in this sentence refers to a restriction that is solidly enforced by the compiler.

```

module BILL
public BILLF1, BILLF2, BILLR1
public FN1, RT1 to BILL1
examine BILL1, BILL2, LIBRARY

function BILLF1(...): integer; ... ; endfunction
function BILLF2(...): boolean; ... ; endfunction
routine BILLR1(...); ... ; endroutine
function FN1(...): integer; ... ; endfunction
routine RT1(...); ... ; endroutine

```

- - - - -

```

module BILL1
examine BILL, BILL2, LIBRARY
public FN11, RT11 to BILL

function FN11(...): integer; ... ; endfunction
routine RT11(...); ... ; endroutine

```

- - - - -

```

module BILL2
public ARR, CAP, STRUC, DAT1, DAT2 to BILL, BILL1

declare
( ARR is array[1..10] of integer
  CAP is structure
    ( // Declare an encapsulated data type.
      ...
    )
  STRUC is structure (...) // Ordinary structure.
  DAT1: integer initially 0
  DAT2: ARR initially construct(ARR, [0..10]: 0)
)

```

Figure 6-10: Example of Separate Compilation



being public with no restriction, so they can be accessed by any module. The next two are public with the restriction "to BILL1", so these can be accessed only within the module named BILL1 (and also within BILL, of course). Now note BILL1, which declares a function and a routine which are known only to BILL. Finally, BILL3 declares some types and some initialized data bases that are known only to BILL and BILL1.

Note also the "examine" statements in BILL and BILL1. The code in BILL is to use the entries in BILL1 and BILL2, as well as those in LIBRARY, a library module, so the compiler must be directed to examine the named modules to determine the necessary information. For example, consider compiling BILL and the examination of BILL1. The compiler looks at the "public" directives in BILL1 and determines that FN11 and RT11 are to be known to BILL. It then looks at the rest of the text of BILL1 to determine the necessary type information about their arguments, so as to be able to enforce the COL's strong typing rules. (Certain hard problems are glossed over here: they are addressed in detail in section 6.2.3. That section also points out a more efficient way to accomplish this.)

#### 6.2.2: Syntax for Separate Compilation

The syntax used by the COL to provide communication between separate modules is now presented, followed by discussion of the relevant semantics.

```
286 <module> ::=                                // separate compilation
287     <module head> ; <module body>

288 <module head> ::=                            // module head
289     module <module ID> ; <module head element> ; ...

290 <module head element> ::=                    // module head element
291     examine <module ID> , ...                // look at another module
292     public ID , ...                          // IDs known everywhere
293     public ID , ... to <module ID> , ...

294 <module ID> ::=                             // name of a module
295     ID

296 <module body> ::=                            // the body of a module
297     SD ; ...
```

Note that the name of the module must come first, but that the <module head element>s may be in any order and that there may be multiple instances of each kind. (Recall that the example shows two "public" lines in BILL.)

The first item in the <module head> gives the module the name by which it is referred to in COL text. As mentioned in section 1.3.1, the compiler must be able to retrieve the text of a module from its name.

An "examine" directive directs the compiler to examine those modules named, in the order of the appearance of the names in the directive. The examining process consists of looking at "public" directives in the examined modules, considering only those that are available to the module being compiled. (See the next paragraph.) Declarations for each available ID are then searched for and included in the module being compiled.

The "public" directive names those IDs in a module which are to be known outside of it. Each ID listed must be declared at the top level of the module. The "public...to" option specifies that the IDs are to be known only to the module(s) named and not to others; in the absence of a "to" clause they are known to all modules which examine this module.

The <module body> is a sequence of statements and declarations. Any ID listed in the "public" list must be declared in one of these declarations. That is, it may not be declared in a nested block.

There is a restriction imposed on any public procedure. If it has an argument or returned value which is a named type, then the declaration of that type must also be public.

#### 6.2.3: Notes on the Implementation

The description of the semantics of the "examine" statement presented above is in terms of a proposed implementation. This discussion suggests that the compiler does the examine on the source text of each examined module, a possibly expensive process. In fact, we intend that the examination be performed on the relative binary version, which contains the symbol table information. It is necessary also that adequate information (such as compilation data and time) be included in the symbol table of each compiled module so that the linker can insure the integrity of the type checking. Information must also be included about the

compilation time of each examined module. The remaining details are not presented here.

### 6.3: Machine Dependent Features

One of the COL's design goals has been that the language be sufficiently complete that entire applications can be coded in the COL, with no modules at all being coded in assembler or any other language. Given that many applications of interest in communications run in stand-alone computers (i.e., with no resident operating system), it is necessary that at least a few of the modules be extremely dependent on the fine details of the hardware. Specifically, the following features are required in the COL:

- . generation of specific sequences of machine instructions. The programmer must be able to specify exactly which instructions are compiled. Often specialized instructions are needed that are not usually compiled by most compilers, as for interrupt handling, I/O, memory map management, etc.
- . control (sometimes in detail) of how data are laid out in memory. For example, it may be necessary that certain items all reside in the same page.
- . access to specific memory locations. In many computers I/O is performed by storing into specific memory cells,



and information about I/O status is determined by reading specific cells.

The COL must meet all of these requirements in a graceful manner.

#### 6.3.1: Machine-Like Code

The COL permits the programmer to cause any sequence of instructions he wants to be included as part of his compiled program. The mechanism uses syntax that is normal COL text. Recall that <machine-like code> is a possible type of statement, as mentioned on line 138 on page 90. We have the following syntax:

```
298 <machine-like code> ::=
299     code <computer ID> do SD ; ... endcode
300 <computer ID> ::=
301     ID                                // which computer?
--- SD ::=
    (see section Ch3)
```

The ID that appears after "code" specifies which computer is being compiled for. The compiler of course knows which computer it is currently compiling for and so flags as an error any code bracket that specifies a different computer. The code within code brackets is by definition not portable and must be examined by a programmer before it can be expected to run on another computer. This restriction provides a check that such examination and subsequent editing is performed.



Although the syntax for "code...endcode" and "begin...end" seems similar, there is an important difference: Within code brackets many additional names are automatically declared to give the programmer direct access to the hardware. Access to specific machine operations is through certain built-in functions and extra operators that are available to the programmer within code brackets. This description provides little insight into how the mechanism is used or what machine code in the COL looks like, since such understanding can be imparted only in terms of some specific object machine. Therefore there appears in Appendix II a description of the Lockheed SUE computer in terms of COL, with some examples of use.

#### 6.3.2: Data Layout in Memory

The <data declaration> is mentioned as a possible <declaration> in the syntax in section Ch2, but the definition of this class has been postponed to this point.

```
302 <data declaration> ::=
303     <storage> data ( ID , ... )
304     <storage> data <memory descriptor> ( ID , ... )

--- <storage> ::=
    (see section 2.2.1)

305 <memory descriptor> ::=
306     ID
```

A data declaration specifies where data are to be stored in memory. In the first kind in line 303, each of the IDs is to be stored consecutively in the order listed. The second kind in line

304 provides an additional piece of information, the <memory descriptor>. This is a machine- or implementation-dependent piece of information about where the data are to go. For example, the descriptor "here" specifies that the data are to be stored in line at the present value of the instruction counter. Another example: it might specify which page is to be used, so that all data relevant to a certain part of the program could be stored on the same page. Since this is so dependent on the details of each implementation, nothing further can be added here.

The class <storage> used in this syntax may be used to specify an exact memory location for the data. The only <storage> that may be used in this context is the "location" attribute from line 45.

In line 46 in section 2.2.1 (page 49) it is suggested that the programmer can specify that a variable is to reside in a hardware register. Left undefined is the syntax for <register>. Although the details depend on the machine, a typical implementation might be this:

```
<register> ::=  
  NC  
  any  
  ID
```

The first line is used to specify that the variable is to reside in the indicated register number, the second line that the

variable is to reside in any register of the compiler's choosing. The third line might be used to specify that any of a class of registers may be used. For example, in the IBM 370 "E" might be used in this context to specify any floating point register.

### 6.3.3: Absolute Addressing

In some cases it is necessary that the programmer be able to refer to specific memory addresses. For example, input/output in some hardware architectures is performed by storing appropriate bit patterns into certain memory locations and the status of the I/O determined by reading memory. The "location" attribute may be used as a <storage> in declaring a variable, as provided for in line 45 in section 2.2.1. For example, the declaration

```
declare (CLOCK: volatile integer location (16!C000))
```

declares the variable CLOCK of type integer and specifies the memory location for the variable. The attribute "volatile" forces the compiler to access this location each time CLOCK appears in the code, clearly necessary if the variable is a clock.

Within code brackets, the programmer may use the function "ref", which returns the address of its argument formatted as a COL pointer. Thus "ref" is a sort of inverse of "@".

To specify into what memory locations the compiled code is to go, the "%linker" compiler directive may be used.

## 6.4: Assertions

Assertions provide a mechanism that permits the user to give the compiler certain information. The syntax is simple:

```
307 <assertion> ::=  
308     assert E           // E is true
```

Here E is a boolean expression, and the user is claiming that E is true at that point in the program. There are three possible actions that the compiler can take on encountering an assertion, depending on compiler directives then in effect:

- . It can compile a check that the assertion is really true. This mode is especially useful in debugging. A run time diagnostic is provided if the assertion fails. (Use of this features requires that an error handling package be available to the running program. See section 6.5.)
- . It can take the programmer's word that the assertion is true, taking advantage of it (to the extent possible) in code optimization.
- . It can ignore it entirely.

Even if the assertion is ignored, its presence acts as a comment to help the self-documentation of the code. Note that it is meaningful to have both of the first two in effect simultaneously.

Default in the absence of any other specification is to ignore assertions. The controlling compiler directive is described in section 5.3.2; it is the "%assertions" directive.

It is the intent of the COL design that the presence of an assertion is not to change the semantics of the program. It is therefore bad coding practice for the programmer, using "%assertions check" mode, to write an assertion which invokes a function with a side effect. Although we recommend strongly that this practice be refrained from, we do not choose to rule it out of the language. There is no way to keep the programmer from writing such a thing without crippling the power of the language in some serious way.

#### 6.5: Run-Time Library

A design principle followed in writing the COL (see section 1.1.3) has been that the programmer should get a lot of code executed only if he writes a lot. An implication of this is that innocent looking statements should not compile into invocations of expensive library procedures. Consistent with this philosophy, the COL has only a few requirements for a run time library of language related procedures. However, procedures are required in connection with the "allocate" function and the "free" statement, for type conversion invoked by the "convert" function, and for a run time error reporting package. These are now described.



Section 4.3 provides syntax for an "allocate" function that has as its first "argument" a <type>. The compiler, knowing the amount of memory required to hold a datum of that <type>, compiles code to invoke a run time function that returns a pointer to a space of that size. If the programmer has included initialization as part of his "allocate" call, appropriate code is compiled to perform it. Consider for example the declaration in Figure 6-11.

```
-----  
declare  
  ( Node is structure  
    ( Item: integer  
      Next: pointer Node  
    )  
    P: pointer Node  
  )
```

Figure 6-11: Sample Structure

-----  
(This is part of a larger program in section 7.5.) Given this declaration, the assignment statement

```
P := allocate(Node, Next: nil)
```

compiles somewhat as if the programmer had written

```
P := ALLOCATE (2)    // Assume Node has size 2  
P@.Next := nil
```

The first statement(40) returns a pointer to the requested space, and the second sets the named field. There is no assumption that the programmer can safely make regarding the value of P@.Item -- it holds whatever bit pattern happens to reside in that memory location at the time.

In general, the value of a declared constant must be known at compile time. However, a constant of type pointer may be initialized by a call to "allocate". The allocation is done as part of the linking operation and the space thus allocated may not be freed. Consider for example the declaration

```
declare(P = allocate(pointer integer: 0))
```

(This uses the syntax on line 242.) As part of the linking process, a memory space large enough to hold an integer is initialized to zero and the constant P has as value the L-value of that space.

Note that the COL's strong typing precludes the possibility invoking ALLOCATE\_ directly, since it is not possible to describe the <type> it returns. (This is the problem alluded to in footnote 40.) As suggested below, the compiler acts as if the returned type is "pointer general", so one may write

```
P := force(pointer NODE: ALLOCATE_(2))
```

to achieve the effect.

---

(40) There is a problem being glossed over here which is returned to later.

In order that the free space pool not be exhausted during execution of the program, it is necessary that allocated space be returned. (There is nothing automatic in the COL about this.) The COL statement

```
free(P, Q, R)
```

returns to free space the memory pointed to by the pointers P, Q and R. It is an error detected at run time if these pointers do not hold values returned by "allocate". The statement compiles into an invocation of the polyadic routine FREE\_, each of whose parameters is a structure holding the pointer to be freed and an integer which is the size of the space.

The procedures ALLOCATE\_ and FREE\_ must be available at run time. Further, the compiler must know about them at compile time. Each module compiled is treated as if it includes

```
examine FREE_PACKAGE
```

in its header. (This syntax is described in section 6.2.2.) The module FREE\_PACKAGE is shown in outline form in Figure 6-12. Note that the descriptor of FREE\_'s argument type is also public.

A second function for which run time procedures may be required in many implementations is conversion between one type and another. Section 6.1.6 lists the conversions that the programmer may invoke using the "convert" function. To the extent practical the compiler does these conversions with in line code, but in some architectures subroutine calls are more appropriate.

```

module FREE_PACKAGE
public ALLOCATE_, FREE_, FREE_ARG_

declare
( FREE_ARG_ is structure
  ( P: pointer general
    N: integer
  )
)

function ALLOCATE_(N: integer): pointer general
  // Find free space of size N and return a pointer to it.
  ...
  // The pointer is in local variable P.
  result is P
endfunction

routine FREE_(variadic A: array[1..?K] of FREE_ARG_)
  declare
  ( P: pointer general
    N: integer
  )
  for J := 1 to K do
    P := A[J].P
    N := A[J].N
    // Return to free space N words at P.
    ...
  endfor
endroutine

```

Figure 6-12: The Module FREE\_PACKAGE

-----

Further details are not presented here.

One final run time action is required: error reporting. Although the COL design has concentrated on making as many errors as possible detectable at compile time, there are some errors that are detectable at run time only. Examples include the following:

- . a "fail" statement with no "failing" trap set.
- . an assertion check that fails (if this feature is turned on by the "%assertion check" compiler directive).
- . a subrange check that fails (if this feature is turned on by the "%check subrange" compiler directive).
- . an "allocate" with insufficient space available.
- . a "free" of a pointer not returned by "allocate".

Although the remaining details are implementation dependent and therefore beyond the scope of this project, it is important to keep in mind that this problem must be solved.



## Chapter 7: Examples

One studying a new computer language, particularly one as complex as the COL, requires many examples of correct COL coding. Table 7-1 lists the more interesting examples that have appeared throughout the earlier chapters. Section 1.2 contains several simple examples with discussion, and this section contains several more completely worked out examples. There are further examples in Appendix II, but these latter are dependent on the COL as it might be extended for a particular computer -- the Lockheed SUE.

## 7.1: Simple Examples

Section 2.5 contains several examples of simple COL declarations.

The following example shows the use of declared types. First we declare some types:

```
declare
( XTYP is integer
  XRange is [1..20]
  XVEC is array XRange of XTYP
  A, B, C: XVEC
)
```

Here XTYP is a synonym for integer and XRange is a subrange from 1 to 20; changing either of these would change the effect of the rest of the declarations. Each of A, B and C is an array from 1

Re.

Figure	Page	Title
1-1	19	Example: Matrix Multiply
1-2	20	Example: Interchange Sort
1-3	21	CRC Checksum
2-1	58	Adding Apples and Oranges
2-2	69	Example of a Structure Declaration
2-3	71	Host to Host Message Format in ARPANET
2-6	87	Declaring Mutually Recursive Procedures
2-7	88	Use of "undeclare"
3-1	97	Sample Conditional Statement
3-12	114	Example of Zahn's Device
3-13	116	Equivalence for Zahn's Device
4-1	137	Examples of COL Precedence
4-2	152	Example of Aggregate Usage
6-1	187	The Variadic Function MaxI
6-2	192	Binding of Free Variables in Types
6-3	194	Scope in Open Procedures
6-4	196	Example of Macro
6-5	199	An Encapsulated Data Type
6-6	204	Example of the "overflow" Event
6-7	207	Zahn's Device and "overflow"
6-8	211	Capturing a "fail" for Finalization
6-9	212	Coercion of "different" Types
6-10	221	Example of Separate Compilation
6-12	235	The Module FREE_PACKAGE

Table 7-1: Examples of COL Usage

- - - - -

to 20 of integers. Now we declare a function on XTYPs that returns the maximum of two of them:

```
function XMAX(S, T: XTYP): XTYP
  result is when S > T then S else T
endfunction
```

Note that the arguments are called by read only. Finally, we define a function on XVECs that returns an XVEC each of whose elements is the maximum of the corresponding elements of its

arguments:

```
function XVMAX(P, Q: XVEC): XVEC
  declare ( T: XVEC )
  for N in XRANGE do T[N] := XMAX(P[N], Q[N]) endfor
  result is T
endfunction
```

Of course, these two function definitions must appear within the scope of the type definitions shown above. Note that had XTYP been declared, say, float, the entire example would still be correct but would define instead the function XVMAX on arrays of floating point numbers.

#### 7.2: Formatted Output: a Variadic Procedure

The next example shows an extremely simple routine PRINT for formatted output. This routine is too simple to be really useful in practice, but it suggests a technique that could in fact be used. It takes an arbitrary number of arguments, the first of which is a character string that controls the format of what is to be printed. The routine maintains a counter Sn through this string S, as well as a counter Vn through the remainder of the arguments, each of which is a pointer to general. String S is scanned looking for the character "%", all other characters being printed immediately. When "%" is found, the character immediately following it is examined to determine how to treat the next parameter to PRINT. If that character is "I" or "i", the next parameter is treated as an integer, etc. The code is shown in Figure 7-1. The assumption is that PUTC is a routine that sends a

```

routine PRINT
( S: array[1..?S_MAX] of char,    // format string
  variadic V: array[1..?V_MAX] of general
)

macro SEND(F, T) = "
  Vn := Vn + 1
  if Vn > V_MAX do Error() endif // report an error...
  F(force(T: V[Vn]))
  stopswitch"

declare
( Sn: integer initially 0    // count through S
  Vn: integer initially 0    // count through V
)

while Sn < S_MAX do
  Sn := Sn + 1    // Next char of S.
  if S[Sn] <> %% do PUTC(S[Sn]); loop endif
  Sn := Sn + 1
  switchon S[Sn] into
    case $I: case $i:    // integer
      SEND(PUTI, integer)
    case $L: case $l:    // logical
      SEND(PUTL, logical)
    case $B: case $b:    // boolean
      SEND(PUTB, boolean)
    case $F: case $f:    // floating point
      SEND(PUTF, float)
    default:
      // report an error...
  endswitch
endwhile
endroutine

```

Figure 7-1: PRINT: a Variadic Procedure

-----

character to the output device, PUTI one that sends an integer, PUTL a logical, and PUTB a boolean. Note the use of the macro SEND to save writing essentially the same piece of code three

times. Its parameters are the routine to call and the type to be used as an argument to force. The latter (see section 4.4) forces the compiler to use the type indicated for the argument. Note that the routine is not very robust. (Another way to put it is that it has a bug.) It will fail in an unpredictable way if its first argument is an empty string, since `S[1]` will be accessed although it does not exist.

The routine is easy to use. For example, the code

```
PRINT("There are %I elements.", W)
```

would cause the following to be printed, assuming that `W` is an integer variable with value 17:

```
There are 17 elements.
```

Here the formal parameter `S` is bound to an array from 1 to 22 of characters, `S_MAX` to 22, `V_MAX` to 1, and `V[1]` to a pointer to the integer variable `W`. As another example, consider the call

```
PRINT("M = %I, N = %I, P = %I", M, N, P)
```

Here `V_MAX` is bound to 3 and the elements of `V` to pointers to the three variables indicated. If `M` has the value 5, `N` the value -14, and `P` the value 127, then the line

```
M = 5, N = -14, P = 127
```

would be printed. The fragment of code

```
M := 1
for N := 1 to 10 do
  M := M*N
  PRINT("%I factorial = %I*C*L", N, M)
endfor
```

would print on 10 successive lines the factorial of the first ten



integers. Note the use of \*C for carrier return and \*L for line-feed in the string. The assumption is the usual one in ASCII that both are needed for transition from one line to the next.

As another example, consider a sparse M by N array, most of whose elements are zero. The code fragment

```
for I := 1 to M do
  for J := 1 to N do
    if V[I, J] ne 0.0 do
      PRINT("%I*T%I*T%F*C*L", I, J, V[I, J])
    endif
  endfor
endfor
```

causes all non-zero elements to be printed, each on a separate line preceded by the row and column number. Here tab characters, represented in the format string by \*T, have been used for spacing. This example points out that it is not necessary that all the arguments after the string be the same type.

To make the PRINT routine really useful, all that is needed is to provide more format control. For example, "%4N" might be used in the format string to specify that an integer is to be printed in a 4-column field. Control of leading blanks or leading zeros, more detailed control of floating point format, etc., all could readily be added.

### 7.3: Interlocks and Zahn's Device

The next example shows the use of interlocks and of Zahn's device. It merely sketches the outline of the program, omitting

all of the code that does the work. Assume that a FUM is some sort of resource, and that there are FUM\_N of them (a constant known at compile time). The array FUM\_LOCK declared by

```
declare ( FUM_LOCK: array[1..FUM_N] of interlock )
```

provides an interlock for each FUM. The code to be written loops through this array, looking for an unlocked FUM, and locks it. It performs some operation, then unlocks the interlock and is done. If no unlocked FUM is found something else is done. An iteration statement loops through the interlocks, and failure to find a free FUM is indicated by completing the iteration. Zahn's device provides a convenient structured way to program the needed branch at the bottom to execute different code depending on whether or not a FUM was found. Recall the semantics of the region statement with the iflocked option (see section 3.4.3): If the interlock was previously set, the code after "iflocked" is obeyed. In the present case it merely loops to the next item of the iteration statement. If the interlock is found unlocked, it is locked and the code after "otherwise" is obeyed. The code is shown in Figure 7-2. Lines containing "..." indicate the omission of code irrelevant to this example.

#### 7.4: Binary Trees

The following example deals with binary trees, used to store a collection of integers so that they are kept ordered. It is assumed that it is necessary to keep track of multiple occurrences

```

upon FOUND or NOT_FOUND leave // Zahn's device
// initialization stuff...
...
for N in [1..FUM_N] do
    ...
    region FUM_LOCK[N] // Test the Nth interlock.
    iflocked // Already locked.
        loop // to get the next value of N
    otherwise // Hah! Found one!
        // Here with the interlock locked.
    ...
    endregion
    // Here having done the task.
    signal FOUND
endfor
// Here on completion of the "for" statement.
signal NOT_FOUND
do // Now the rest of the Zahn statement.
    case FOUND: // Here after signaling FOUND.
        ...
        stopswitch
    case NOT_FOUND:
        // Here if no unlocked FUM was found.
        ...
endupon

```

Figure 7-2: Example: Zahn's Device

-----

of the same number. Each node of the tree has an entry which contains both the integer and a count of the number of instances of that integer. It also has a left child and a right child, each of which is either empty (represented by a nil pointer) or is a pointer to a tree. All entries found in the left child and its descendants are numerically less than that at the node, and all in the right are greater. To deal with such trees we first declare a

new type for this structure:

```
declare
  ( Tree is
    structure
      ( Entry: integer
        Count: integer
        Left: pointer Tree
        Right: pointer Tree
      )
    )
```

Note that each of Left and Right are pointers. If a node does not have, say, a left child, then the entry in that field is nil. (Recall from section 4.1 that nil is an expression denoting a pointer to nothing. It passes the type checking rules as being a pointer, but it is an error to attempt to point through it. It can be checked for equality, as seen below in the routine Walk.)

Now we define a routine Enter to make an entry in the tree, shown in Figure 7-3. If the number to be entered is not yet there, it is added; while if it is already present its count is incremented. This simple recursive routine solves the problem. Note that its argument is a pointer to a tree, since it might be nil. The array is called by reference since the corresponding actual parameter is to be updated if its previous value is nil.

Now we want a tree walking routine to examine all the nodes of the tree. One parameter to the walk routine is a routine that takes two integer arguments: the entry and the count at a node. Walk calls this routine for each node in the tree, in increasing order by the entry. It is convenient to have a type for this

```

routine Enter(ref T: pointer Tree, N: integer)
  if T = nil do // Nothing at this node.
    T := allocate(Tree, Entry:N, Count:1, Left:nil, Right:nil)
    return
  endif
  // We have an entry. Look at it.
  test N = T@.Entry do // The node we want?
    T@.Count := T@.Count + 1 // Increment.
  orif N < T@.Entry do
    Enter(T@.Left, N)
  otherwise
    Enter(T@.Right, N)
  endtest
endroutine

```

Figure 7-3: Example: Tree Walking

```

-----

routine:
  declare
    ( ZAP is routine
      ( Entry: integer,
        Count: integer
      )
    )

```

It is now easy to write Walk; the code is shown in Figure 7-4.

Finally, we build a tree with some entries, as shown in Figure 7-5. The last statement exercises all of this machinery and causes 6 to be stored into N. Note that N is in static storage so that it can be accessed within the body of F. To count the total number of entries, it would be necessary only to change the body of F to read

```
N := N + C
```



```
routine Walk(T: pointer Tree, FOO: ZAP)
  unless T = nil do
    Walk(T@.Left, FOO)
    FOO(T@.Entry, T@.Count)
    Walk(T@.Right, FOO)
  endunless
endroutine
```

Figure 7-4: Example: Routine for Walk

-----

```
declare (Root: Tree) // A Tree.

// Build a tree.
Root := nil
macro F(n) = "Enter(Root, n)"
F(1); F(3); F(-1); F(2); F(4); F(1); F(5)

// To count the number of distinct nodes...
declare(N: static integer)

routine F(E: integer, C: integer)
  N := N + 1
endroutine

N := 0
Walk(Root, F)
```

Figure 7-5: Example: Use of Walk

### 7.5: An Encapsulated Type

Figure 7-6 shows an example of a capsule which provides a safe implementation of queues, the safety coming from the fact that there is no way for the user of such a queue to get at the storage used.

A queue is represented in this example by a linked list of NODEs, in which each NODE holds an item on the queue (an integer) and a pointer to the next NODE. Each queue has three fields, inaccessible to the outside of the capsule. (The entire capsule is declared to be private and the fields are not marked "public".) The field HEAD contains a pointer to the head of the linked list -- the next item to be returned. The field TAIL points to the last item on the list, the one after which to store the next item to be enqueued. The field Q\_LOCK is an interlock. The assumption is that this module is used in a multiprocessor environment, in which it is possible that more than one processor is attempting simultaneously to use the code in this capsule. Since chaos would result were two processes trying to change the pointers here, an interlock is used to prevent this from happening. This point is returned to below.

Three cases must be distinguished in describing the representation: an empty queue, a queue containing exactly one entry, and a larger queue. An empty queue is represented by HEAD being nil. (The value of TAIL is not relevant in this case.) If

```

declare
  ( DQ_RESULT is structure  // The type returned by DEQUEUE.
    ( ITEM: integer
      EMPTY: boolean
    )

    QUEUE is private structure  // This is the capsule.
    ( declare
      ( NODE is structure
        ( ITEM: integer
          NEXT: pointer NODE
        )
      )

      HEAD: pointer NODE initially nil
      TAIL: pointer NODE
      Q_LOCK: interlock initially unlocked

      public routine ENQUEUE(N: integer)
      declare
        ( V: pointer NODE
          initially allocate(NODE, ITEM: N, NEXT: nil)
        )
        region Q_LOCK do
          (when HEAD=nil then HEAD else TAIL@.NEXT) := V
          TAIL := V
        endregion
      endroutine

      public function DEQUEUE(): DQ_RESULT
      if HEAD=nil do
        result is construct(DQ_RESULT, EMPTY: true)
      endif
      declare(N: integer; V: pointer NODE)
      region Q_LOCK do
        N := HEAD@.ITEM
        V := HEAD
        HEAD := HEAD@.NEXT
      endregion
      free(V)
      result is construct(DQ_RESULT, ITEM: N, EMPTY: false)
    endfunction

    // There is more of this program on the next page...

```

```

        finish // Finalization code.
            declare (V: pointer NODE)
            while HEAD ne nil do
                V := HEAD
                HEAD := HEAD@.NEXT
                free(V)
            endwhile
        endfinish
    )
)

```

Figure 7-6: QUEUE: An Example of a Capsule

-----

there is exactly one item on the queue, both HEAD and TAIL point to it. The NEXT pointer in that node is nil. For a larger queue, HEAD points to the first NODE, whose NEXT points to the next NODE, ..., whose NEXT points to the last NODE. This last NODE is also pointed to by TAIL and has a NEXT pointer which is nil.

A few comments about the usage of the COL in this example may help the reader. Note that ITEM is used as a field name in two structures, DQ\_RESULT and NODE. There is no problem from this, since full qualification is required. (That is, the programmer may not write ITEM other than as a component of a structure which he names.) The capsule QUEUE is a "private" structure. This makes the default for all of its entries private, so that the only public entries are those explicitly so labeled. In this case only

ENQUEUE and DEQUEUE are public. The fields HEAD and Q\_LOCK have the initial attribute. This means that declaring a variable of type QUEUE causes these fields to be initialized as part of the declaration. Such initialization is clearly required before usage of this capsule, in order to insure the consistency of the pointer structure. That is, the sequence

```
declare (Q1: QUEUE)
Q1.ENQUEUE(5)
```

first declares Q1 to be a queue and then places 5 on that queue. The initialization of Q1.HEAD to "nil" that takes place as part of the declaration of Q1 permits the ENQUEUE operation to do the right thing. No other convenient way to initialize Q1.HEAD is available(41), since the privacy of Q1.HEAD means that the programmer may not initialize it himself after declaration of Q1. It is this effect that makes this implementations of queues safe, since there is no way for the programmer to alter the pointer structure so as to render it inconsistent.

Note that the critical part of each of ENQUEUE and DEQUEUE is a region using the interlock Q\_LOCK, thus insuring that it is not possible for more than one processor at a time to be executing in such a region. Each region is as short as possible, containing no statements that could be safely moved outside of the region. In particular, the "allocate" and the "free" are performed outside of

---

(41) Of course, a "start" field could be used.



the regions, since they might well be rather time consuming.

Finally, note the "finish" code (on the second page of the figure). If a normal exit is made from a block in which a queue is declared, it is desirable that the space in use to hold the queue be returned to the free space pool. The finish code does this.

#### 7.6: Quick Sort

A final example is of a quick sort routine, shown in Figure 7-7. This program appears on page 80 of [Wirth-76], accompanied by a description of the algorithm. The program has been transcribed from PASCAL to the COL, a relatively simple task. Since the code is described in the source, no further discussion is necessary.

```

declare
(  ITEM is structure
  (  KEY: integer    //  sort on this key
    CONTENTS: integer
  )
)

routine QUICKSORT(ref A: array[1..?N] of ITEM)
  declare
    (  M = 12    //  stack size
      KEY: integer    //  temp for a key
      I, J, L, R: integer    //  some temps
      S: [0..M] initially 1    //  stack counter (last used)
      STACK:    //  stack of intervals to be looked at
                array [1..M] of structure (L: integer; R: integer)
    )

    STACK[1].L := 1    //  Stack the entire interval.
    STACK[1].R := N
    repeat //  until the stack is empty
      L := STACK[S].L    //  L, R get top stack entry
      R := STACK[S].R
      S := S - 1    //  remove it from the stack
      repeat
        I := L
        J := R
        KEY := A[(L+R)/2].KEY
        repeat
          while A[I].KEY < KEY do I := I + 1 endwhile
          while KEY < A[J].KEY do J := J - 1 endwhile
          if I <= J do
            swap(A[I], A[J])
            I := I + 1
            J := J - 1
          endif
        until I > J
        if I < R do
          S := S + 1
          STACK[S].L := I
          STACK[S].R := R
        endif
        R := J
      until L >= R
    until S = 0
  endroutine

```

Figure 7-7: Quick Sort Routine

## Bibliographic References

Many languages are referred to in the body of this document without providing references to them. Citations for them are found below as follows:

ALGOL-60	[Naur-63]
ALPHARD	[Wulf-76]
BCPL	[Richards-69], [BBN-74]
BLISS	[Wulf-71]
CS-4	[Intermetrics-75]
EUCLID	[Lampson-77]
FORTTRAN	[FORTTRAN-76]
IMP	[Irons-70]
JOVIAL	[Shaw-63]
PASCAL	[Jensen-74]
PL/I	[IBM]
SIMULA	[Dahl-70]

[BBN-74]

"BCPL Manual", Bolt Beranek and Newman Inc, Sep 74

[Brinch-Hansen-72]

Per Brinch Hansen, "Structured Multiprogramming", Comm ACM 15(7) Jul 72, pp 574-578

[Brinch-Hansen-73]

Per Brinch-Hansen, "Operating Systems Principles", Prentice-Hall, Inc, 1973

[Dahl-70]

Ole-Johan Dahl, Bjorn Myhrhaug, Kirsten Nygaard, "Common base language", Norwegian Computing Center, Publication S-22, Oct 70

[DoD-77]

"Department of Defense requirements for high order computer programming language -- Ironman," 14 Jan 77

[Evans-76]

Arthur Evans Jr, C Robert Morgan, "Development of a communications oriented language", BBN Report No 3261, 20 Mar 76

[Fisher-76]

D A Fisher, "A common programming language for the Department of Defense -- background and technical requirements", Paper P-1191, Institute for Defense Analysis, Jun 76

[FORTRAN-76]

"Draft proposed ANS FORTRAN", SIGPLAN Notices 11(3) Mar 76, entire issue

[IBM]

"PL/I language specification", IBM Corporation, Form GY33-6003-2, undated

[Intermetrics-75]

"CS-4 language reference manual and operating system interface", Intermetrics, Inc, IR-130-2, Oct 75

[Irons-70]

E T Irons, "Experience with an extensible language", Comm ACM 13(1) Jan 70

[Jensen-74]

Kathleen Jensen, Niklaus Wirth, "PASCAL user manual and report", second edition, Springer-Verlag, 1974

[Knuth-73]

Donald E Knuth, "A review of structured programming", STAN-CS-73-371, Computer Science Department, Stanford University, Jun 73

[Knuth-74]

Donald E Knuth, "Structured programming with goto statements", Computing Surveys, Dec 74

[Lampson-77]

B W Lampson, J J Horning, R L London, J G Mitchel, G J Popek, "Report on the programming language EUCLID", SIGPLAN Notices, 12(2) Feb 77, entire issue

[Morgan-77]

C Robert Morgan, Arthur Evans Jr, "Communications oriented language (COL): language implementation and usage", BBN Report No 3533, 2 May 77

[Naur-63]

Peter Naur, editor, "Revised report on the algorithmic language ALGOL 60", Comm ACM Jan 63, pp 1-17

[Richards-69]

Martin Richards, "BCPL -- a tool for compiler writing and systems programming", SJCC 1969, pp 557-566

[Shaw-63]

Christopher J Shaw, "A specification of JOVIAL", Comm ACM 6(12) Dec 63, pp 721-736

[Wirth-76]

Niklaus Wirth, "Algorithms + data structures = programs", Prentice-Hall, Inc, 1976

[Wulf-71]

W A Wulf, D B Russell, A N Haberman, "BLISS: a language for system programming", Comm ACM 14(12) Dec 71, pp 780-790

[Wulf-76]

W A Wulf, Ralph L London, Mary Shaw, "Abstraction and verification in ALPHARD: introduction to language and methodology", Department of Computer Science, Carnegie-Mellon University, 14 Jun 76

[Zahn-74]

Charles T Zahn, "A control structure for natural top down structured programming", Symposium on programming languages, Paris, 1974



## Appendix I: Summary of Syntax

This appendix contains two tables to assist the reader in finding his way around the COL's syntax. The first table lists each syntactic class, showing for each the line number and page number where it is defined in the text. The second table lists the COL's complete syntax, also keyed by line and page number to the text.

## Ap1.1: Syntactic Classes

Following is a table showing for each syntactic class the line number on which it is defined and the page on which it appears.

Syntactic Class	Line	Page	Section
<declaration> . . . . .	1	38	Ch2
<undeclaration> . . . . .	9	39	Ch2
<function declaration> . . . . .	11	41	2.1
<routine declaration> . . . . .	14	41	2.1
<mode> . . . . .	17	41	2.1
<fpl> . . . . .	21	41	2.1
<fp> . . . . .	24	41	2.1
<call type> . . . . .	26	41	2.1
<scalar declaration> . . . . .	31	48	2.2
<decl> . . . . .	33	48	2.2
<variable decl> . . . . .	37	49	2.2.1
<volatility> . . . . .	39	49	2.2.1
<storage> . . . . .	42	49	2.2.1

Syntactic Class	Line	Page	Section
<init value> . . . . .	48	49	2.2.1
<constant decl> . . . . .	51	54	2.2.2
<type decl> . . . . .	54	56	2.2.3
<event declaration> . . . . .	57	59	2.3
<event> . . . . .	59	59	2.3
<type> . . . . .	61	60	2.4
<unsized type> . . . . .	63	60	2.4
<size> . . . . .	67	61	2.4.1
<basic type> . . . . .	75	63	2.4.2
<charset> . . . . .	84	63	2.4.2
<aggregate type> . . . . .	87	65	2.4.3
<array bound> . . . . .	93	65	2.4.3
<field list> . . . . .	96	65	2.4.3
<field> . . . . .	98	65	2.4.3
<var-field> . . . . .	106	66	2.4.3
<case label> . . . . .	108	66	2.4.3
<s-mode> . . . . .	110	66	2.4.3
<other type> . . . . .	114	73	2.4.4
<discrete type> . . . . .	121	73	2.4.4
<limit> . . . . .	124	73	2.4.4
<macro declaration> . . . . .	127	79	2.6
<macro param> . . . . .	130	79	2.6
S . . . . .	132	90	Ch3
<sequence> . . . . .	140	90	Ch3
SD . . . . .	143	90	Ch3
<simple statement> . . . . .	146	92	3.1
<conditional statement> . . . . .	151	95	3.2
<alt list> . . . . .	156	95	3.2
<iteration statement> . . . . .	159	97	3.3
<for list element> . . . . .	165	98	3.3
<for variable> . . . . .	171	98	3.3
<other statement> . . . . .	174	109	3.4
<switch statements> . . . . .	186	112	3.4.1
<case> . . . . .	190	112	3.4.1
<Zahn statements> . . . . .	194	114	3.4.2
<interlock statements> . . . . .	197	118	3.4.3
<failure statements> . . . . .	204	122	3.4.4
E . . . . .	208	130	Ch4
NL . . . . .	213	130	Ch4
NC . . . . .	215	130	Ch4

Syntactic Class	Line	Page	Section
<primary expression> . . .	217	131	4.1
<operator expression> . . .	229	133	4.2
<aggregate expression> . . .	232	147	4.3
<aggregate init> . . . . .	236	147	4.3
<aggregate value> . . . . .	240	147	4.3
<field value> . . . . .	244	148	4.3
<field label list> . . . . .	247	148	4.3
<field label> . . . . .	251	148	4.3
<subscript list> . . . . .	254	148	4.3
<subscript item> . . . . .	256	148	4.3
<subscript> . . . . .	258	148	4.3
<other expression> . . . . .	261	153	4.4
<macro invocation> . . . . .	281	169	5.2
<macro arg> . . . . .	284	170	5.2
<module> . . . . .	286	223	6.2.2
<module head> . . . . .	288	223	6.2.2
<module head element> . . . . .	290	223	6.2.2
<module ID> . . . . .	294	223	6.2.2
<module body> . . . . .	296	223	6.2.2
<machine-like code> . . . . .	298	226	6.3.1
<computer ID> . . . . .	300	226	6.3.1
<data declaration> . . . . .	302	227	6.3.2
<memory descriptor> . . . . .	305	227	6.3.2
<assertion> . . . . .	307	230	6.4

## Ap1.2: Listing of the COL's Complete Syntax

Following is a complete listing of all of the COL's syntax. The line numbers on the left match those in the main body of the document, and page numbers are given on the right.

Line		Page
Ch2:	Declarations	36
1	<declaration> ::=	38
2	<function declaration>	
3	<routine declaration>	
4	<scalar declaration>	
5	<event declaration>	
6	<macro declaration>	
7	<data declaration>	
8	<undeclaration>	
9	<undeclaration> ::=	39
10	undeclare ( ID , ... )	
Section 2.1:	Function and Routine Declarations	40
11	<function declaration> ::=	41
12	<mode> function ID <fpl> : <storage> <type> ; SD ; ...	
	endfunction	
13	forward function ID <fpl> : <storage> <type>	
14	<routine declaration> ::=	41
15	<mode> routine ID <fpl> ; SD ; ... endroutine	
16	forward routine ID <fpl>	
17	<mode> ::=	41
18	open	
19	closed	
20	<empty>	
21	<fpl> ::=	41
22	( <fp> , ... )	
23	( )	

Line	Page
(Section 2.1: Function and Routine Declarations)	
24 <fp> ::=	41
25 <call type> ID , ... : <storage> <type>	
26 <call type> ::=	41
27 value	
28 ref	
29 variadic <call type>	
30 <empty>	
Section 2.2: Scalar Declarations	
31 <scalar declaration> ::=	48
32 declare ( <decl> ; ... )	
33 <decl> ::=	48
34 <variable decl>	
35 <constant decl>	
36 <type decl>	
Section 2.2.1: Declaration of Variables	
37 <variable decl> ::=	49
38 ID , ... : <volatility> <storage> <type> <init value>	
39 <volatility> ::=	49
40 volatile	
41 <empty>	
42 <storage> ::=	49
43 static	
44 dynamic	
45 location ( NL )	
46 register ( <register> )	
47 <empty>	
48 <init value> ::=	49
49 initially E	
50 <empty>	



Line	Page
Section 2.2.2: Declaration of Constant Names	53
51 <constant decl> ::=	54
52 ID , ... = NL	
53 ID , ... = label	
Section 2.2.3: Type Declaration	56
54 <type decl> ::=	56
55 ID , ... is <type>	
56 ID , ... is different <type>	
Section 2.3: Event Declarations	59
57 <event declaration> ::=	59
58 check <event>	
59 <event> ::=	59
60 ID	
Section 2.4: Types	60
61 <type> ::=	60
62 <size> <unsized type>	
63 <unsized type> ::=	60
64 <basic type>	
65 <aggregate type>	
66 <other type>	
Section 2.4.1: Size	61
67 <size> ::=	61
68 <integer> word	
69 <integer> byte	
70 <integer> bit	
71 word	
72 byte	
73 bit	
74 <empty>	

Line	Page
Section 2.4.2: Basic Types	63
75 <basic type> ::=	63
76     integer	
77     float	
78     float ( NC )	
79     logical	
80     <charset> char	
81     boolean	
82     interlock	
83     condition	
84 <charset> ::=	63
85     ID	
86     <empty>	
Section 2.4.3: Aggregates	65
87 <aggregate type> ::=	65
88     array <array bound> , ... of <type>	
89     structure <s-mode> <field list>	
90     packed <aggregate type>	
91     unpacked <aggregate type>	
92     parallel <aggregate type>	
93 <array bound> ::=	65
94     <discrete type>	
95     ID	
96 <field list> ::=	65
97     ( <field> ; ... )	
98 <field> ::=	65
99     <s-mode> ID : <volatility> <type> <init value>	
100     : <type>	
101     selecton ID into ( <var-field> ; ... )	
102     <s-mode> <declaration>	
103     <assertion>	
104     start SD ; ... endstart	
105     finish SD ; ... endfinish	
106 <var-field> ::=	66
107     <case label> <field list>	

Line	Page
(Section 2.4.3: Aggregates)	
108 <case label> ::=	66
109 <case> : ... :	
110 <s-mode> ::=	66
111 public	
112 private	
113 <empty>	
Section 2.4.4: Other Types	
114 <other type> ::=	73
115 pointer <type>	
116 function <fpl> : <storage> <type>	
117 routine <fpl>	
118 general	
119 <discrete type>	
120 ID	
121 <discrete type> ::=	73
122 [ <limit> .. <limit> ]	
123 ( ID , ... )	
124 <limit> ::=	73
125 E	
126 ? ID	
Section 2.6: Macro Declarations	
127 <macro declaration> ::=	79
128 macro ID = <macro body>	
129 macro ID ( <macro param> , ... ) = <macro body>	
130 <macro param> ::=	79
131 ID	
Ch3: Statements	
132 S ::=	90
133 <simple statement>	
134 <conditional statement>	
135 <iteration statement>	
136 <other statement>	
137 <assertion>	

Line	Page
(Ch3: Statements)	
138        <machine-like code>	
139        <sequence>	
140        <sequence> ::=	90
141            begin SD ; ... end	
142            { SD ; ... }	
143        SD ::=	90
144            S	
145            <declaration>	
Section 3.1: Simple Statements	
146        <simple statement> ::=	92
147            E := E	
148            E *= <infix-op> E	
149            E ( E , ... )	
150            E ( )	
Section 3.2: Conditional Statements	
151        <conditional statement> ::=	94
152            if E do S ; ... endif	95
153            unless E do S ; ... endunless	
154            test <alt list> endtest	
155            test <alt list> otherwise S ; ... endtest	
156        <alt list> ::=	95
157            E do S ; ...	
158            E do S ; ... orif <alt list>	
Section 3.3: Iteration Statements	
159        <iteration statement> ::=	97
160            while E do S ; ... endwhile	
161            repeat S ; ... until E	
162            for <for list element> do S ; ... endfor	
163            break	
164            loop	

Line	Page
(Section 3.3: Iteration Statements)	
165 <for list element> ::=	98
166 <for variable> := E step E until E	
167 <for variable> := E incr E to E	
168 <for variable> := E decr E to E	
169 <for variable> := E to E	
170 <for variable> in <discrete type>	
171 <for variable> ::=	98
172 ID	
173 defined ID	
Section 3.4: Other Statements	
174 <other statement> ::=	109
175 goto ID	
176 ID : S	
177 return	
178 result is E	
179 <empty>	
180 swap ( E , ... )	
181 free ( E , ... )	
182 <switch statements>	
183 <Zahn statements>	
184 <interlock statements>	
185 <failure statements>	
Section 3.4.1: The "switchon" Command	
186 <switch statements> ::=	112
187 switchon E into S ; ... endswitch	
188 <case> : S	
189 stopswitch	
190 <case> ::=	112
191 case NC	
192 case NC to NC	
193 default	



Line	Page
Section 3.4.2: Zahn's Device	113
194 <Zahn statements> ::=	114
195     upon ID or ... leave S ; ... do S ; ... endupon	
196     signal ID	
Section 3.4.3: Interlock Statements	117
197 <interlock statements> ::=	118
198     region E do S ; ... endregion	
199     region E iflocked S ; ... otherwise S ; ... endregion	
200     retry	
201     lock E	
202     lock E iflocked S ; ... endlock	
203     unlock E	
Section 3.4.4: The "failure" Mechanism	122
204 <failure statements> ::=	122
205     failing SD ; ... failhere SD ; ... endfail	
206     fail	
207     fail finishing E , ...	
Ch4: Expressions	130
208 E ::=	130
209     <primary expression>	
210     <operator expression>	
211     <aggregate expression>	
212     <other expression>	
213 NL ::=	130
214     E	
215 NC ::=	130
216     E	
Section 4.1: Primary Expressions	131
217 <primary expression> ::=	131
218     ID	
219     <integer>	
220     <floating number>	
221     <character constant>	

Line		Page
	(Section 4.1: Primary Expressions)	
222	<character string>	
223	<logical constant>	
224	true	
225	false	
226	nil	
227	locked	
228	unlocked	
	Section 4.2: Operator Expressions	133
229	<operator expression> ::=	133
230	E <infix op> E	
231	<prefix op> E	
	Section 4.3: Aggregate Expressions	147
232	<aggregate expression> ::=	147
233	E [ E , ... ]	
234	E . ID	
235	<aggregate init>	
236	<aggregate init> ::=	147
237	construct ( <aggregate value> )	
238	allocate ( <aggregate value> )	
239	table ( <type> , E , ... )	
240	<aggregate value> ::=	147
241	<type> , <field value> , ...	
242	<type> : E	
243	<type>	
244	<field value> ::=	148
245	<field label list> : E	
246	<field label list> ( <field value> , ... )	
247	<field label list> ::=	148
248	<field label> . ...	
249	<subscript list> . <field label> . ...	
250	<subscript list>	
251	<field label> ::=	148
252	ID	
253	ID <subscript list>	

Line		Page
	(Section 4.3: Aggregate Expressions)	
254	<subscript list> ::=	148
255	<subscript item> <empty> ...	
256	<subscript item> ::=	148
257	[ <subscript> , ... ]	
258	<subscript> ::=	148
259	NC	
260	NC .. NC	
	Section 4.4: Other Expressions	153
261	<other expression> ::=	153
262	( E )	
263	E ( E , ... )	
264	E ( )	
265	max ( E , ... )	
266	min ( E , ... )	
267	succ ( E )	
268	pred ( E )	
269	abs ( E )	
270	truncate ( E )	
271	round ( E )	
272	floor ( E )	
273	ceiling ( E )	
274	low ( E )	
275	high ( E )	
276	convert ( <type> : E )	
277	force ( <type> : E )	
278	when E then E else E	
279	E @	
280	explicit ( E )	
	Section 5.2: Macros	169
281	<macro invocation> ::=	169
282	ID	
283	ID ( <macro arg> , ... )	
284	<macro arg> ::=	170
285	<any character not comma or right paren> <empty> ...	

Line	Page
Section 6.2.2: Syntax for Separate Compilation	222
286 <module> ::=	223
287     <module head> ; <module body>	
288 <module head> ::=	223
289     module <module ID> ; <module head element> ; ...	
290 <module head element> ::=	223
291     examine <module ID> , ...	
292     public ID , ...	
293     public ID , ... to <module ID> , ...	
294 <module ID> ::=	223
295     ID	
296 <module body> ::=	223
297     SD ; ...	
Section 6.3.1: Machine-Like Code	226
298 <machine-like code> ::=	226
299     code <computer ID> do SD ; ... endcode	
300 <computer ID> ::=	226
301     ID	
Section 6.3.2: Data Layout in Memory	227
302 <data declaration> ::=	227
303     <storage> data ( ID , ... )	
304     <storage> data <memory descriptor> ( ID , ... )	
305 <memory descriptor> ::=	227
306     ID	
Section 6.4: Assertions	230
307 <assertion> ::=	230
308     assert E	

## Appendix II: Machine-Level Code for the SUE

The method for inserting into COL programs suitable statements to cause any specific sequence of machine code to be compiled is introduced in section 6.3. This facility can only be described meaningfully by using as an example some particular target computer. This Appendix therefore contains a description of the Lockheed SUE computer in terms of the COL, followed by a fairly complete example of use of the features defined. Using this material one could program any specific machine instructions for the SUE. The SUE has been selected for two reasons: BBN has extensive experience in programming communications applications on this computer, since it is the processor around which BBN's Pluribus system is built; and the SUE resembles the DEC PDP-11, which is familiar to many.

## Ap2.1: Description of the SUE

The SUE is a mini-computer with the following characteristics:

- . 28,000 16-bit words of memory, byte-addressed
- . simple instructions (no multiply, divide, floating point)
- . 7 general registers, program counter, status register
- . byte or word addressing
- . input output and interrupts via fixed memory locations

Access to machine language from COL is obtained within code



brackets by using variables declared to be specific machine registers, and through a set of special COL subroutines to gain access to special instructions. For example, the instruction to "add the contents of memory location A into register 3" is obtained by first declaring

```
declare (R3_: register(3) integer; A: integer)
and(42) then writing the statement
```

```
R3_ *= + A
```

A similar mechanism is provided for each of SUE's instructions. The idea is that most of SUE's instructions can be represented directly in this manner using COL syntax, and the remaining ones are expressed in functional notation using machine-dependent extensions to the language. Between "code SUE" and "endcode" these names are implicitly declared with the semantics about to be described. The use of "SUE" after "code" indicates that the sequences are specific to the SUE implementation. (See section 6.3.)

The rest of this appendix is organized as follows: First SUE's hardware registers are explained, and then the COL expressions that correspond to SUE's addressing modes. It is then possible to show those COL statements that correspond directly to SUE's instruction set. Next the event identifiers available in

---

(42) The syntax for the register declaration is presented in section Ap2.1.1.

SUE are described. Next, interrupts and I/O in SUE are considered briefly, and finally there are some examples.

A naming convention is used consistently in this appendix for the reader's convenience. All identifiers introduced into the COL that serve to access aspects of the SUE are spelled with uppercase letters with a trailing underscore. It is these names that are available within code brackets. Code written as examples in this Appendix uses uppercase names without a trailing underscore.

One point must be made out of order. The COL expression

`CMP_(A, B)`

where A and B are suitable operands (as described below in section Ap2.1.4) compiles into the SUE compare instruction, which compares A and B and sets certain status bits to indicate the result of the comparison.

#### Ap2.1.1: Changes to the COL for the SUE

Several extensions are made to the COL for use in the compiler for the SUE. The type "index" is similar in many ways to pointer but permits more efficient access in many cases to arrays of words or bytes. In addition, the type "address" is similar to integer, but denotes a 16-bit unsigned integer. (An integer is a 16-bit quantity that uses one bit for the sign.) A size field is not permitted in declaring an index or an address.

The class <register> is used in section 2.2.1 but is not defined elsewhere in this document. For SUE the definition

```
<register> ::=  
    N  
    <empty>
```

is used. Here N if present specifies which register to use; if N is missing the intent is that the compiler is to use any convenient register.

The event IDs defined for SUE are described in section Ap2.1.5.

For SUE the default size for integer and logical quantities is 16 bits, and the size of a byte is 8 bits. Characters are 8 bits wide.

#### Ap2.1.2: Registers in the SUE

The following registers exist in SUE:

0	the program counter
1-7	general registers 1-7
8	the status register.

A declaration for these registers is in Figure Ap2-1. In addition, the 16 bits of the status register may be referred to individually as registers 9 to 24, as shown in Figure Ap2-2. Each of bits 9 to 12 and 16 to 20 is set or cleared by the instructions that affect it. A declaration for these bits is in Figure Ap2-3.

AD-A047 392

BOLT BERANEK AND NEWMAN INC CAMBRIDGE MASS  
COMMUNICATIONS ORIENTED LANGUAGE (COL): LANGUAGE DEFINITION.(U)  
MAY 77 A EVANS, C R MORGAN

F/G 9/2

UNCLASSIFIED

BBN-3534

SBIE-AD-E100007

DCA100-76-C-0051

NL

4 OF 4

ADA047392



END

DATE

FILMED

1 - 78

DDC

```

declare
( PC_ :      register(0) address    // program counter
  R1_ :      register(1) logical    // general register
  R2_ :      register(2) logical    // general register
  R3_ :      register(3) logical    // general register
  R4_ :      register(4) logical    // general register
  R5_ :      register(5) logical    // general register
  R6_ :      register(6) logical    // general register
  R7_ :      register(7) logical    // general register
  STATUS_ :  register(8) logical    // status bits
)

```

Figure Ap2-1: Declarations for SUE Registers

-----

```

9      equal:  A=B in last CMP (A, B)
10     greater than:  A>B in last CMP (A, B)
11     overflow: overflow in last arithmetic operation
12     carry: carry from last arithmetic operation
13-15   flags: 3 programmer-settable bits
16     loopend: set when "INC (A)" or "DEC (A)" results in 0
17     odd: set when arithmetic operation gives odd result
18     zero: set when arithmetic operation gives 0 result
19     negative: set when arithmetic op gives result < 0
20     running; set if the processor is running
21-24   interrupt: bit set means ignore interrupt at level

```

Figure Ap2-2: Bits of the Status Register in SUE

### Ap2.1.3: Addressing Modes

The SUE computer has addressing modes which allow references in a single instruction to registers or memory with indexing and



```

declare // Registers for SUE.
( EQ_: register( 9) boolean // A=B in last CMP_
  GT_: register(10) boolean // A>B in last CMP_
  OV_: register(11) boolean // overflow in last + - shift
  CY_: register(12) boolean // carry in last + - or shift
  F1_: register(13) boolean // programmer's flag 1
  F2_: register(14) boolean // programmer's flag 2
  F3_: register(15) boolean // programmer's flag 3
  LP_: register(16) boolean // 0 in last INC_ or DEC_
  OD_: register(17) boolean // last general operation odd
  ZE_: register(18) boolean // last general op yields zero
  NG_: register(19) boolean // last general op negative
  RN_: register(20) boolean // running
)

```

Figure Ap2-3: Declaration for Status Bits

-----

indirect addressing, and with automatic incrementing or decrementing of an index or pointer as a part of the reference. To provide a succinct notation for the incrementing and decrementing, we provide the two COL functions "INC\_(A)" and "DEC\_(A)" which operate on pointers or indices. The function

INC\_(A)

returns the original A as the value, but as a side effect increments A. If A points to something of size 8 then one is added; if the size is 16, then two is added. The result is that A will then point to the next item in an array of words or bytes. The function

DEC\_(A)

decrements A by one or two as above, returning as value the result

after decrementing.

The addressing modes of the computer are presented using the following variables as examples:

A	an array of objects of size 8 or 16 bits each.
PA	a pointer to such an object, residing in a register.
IA	an index for such an object, residing in a register.
XA	an array of pointers.
PXA	a pointer to such an object, residing in a register.
IXA	an index for such an object, residing in a register.
R	a register.

The declaration shown in Figure Ap2-4 would declare these variables appropriately. Figure Ap2-5 shows by example those COL

```
-----  
  
declare  
( // First declare two types...  
  TYPE_1 is array[0..10] of integer  
  TYPE_2 is array[0..8] of pointer array[0..12] of char  
  
  A:  TYPE_1  
  PA: register pointer TYPE_1  
  IA: register index TYPE_1  
  XA: TYPE_2  
  PXA: pointer TYPE_2  
  IXA: index TYPE_2  
  R:  register logical  
)
```

Figure Ap2-4: Declaration of Sample Variables

expressions that can be accessed as part of the address of a single SUE instruction. These examples use the variables just defined. Any COL expression of the form just presented can be

-----

A	direct addressing of memory.
A[IA]	indexed addressing of memory.
A[INC_(IA)]	index with incrementing of IA.
A[DEC_(IA)]	indexing with decrementing.
R	direct reference to a register.
PA @	reference through a pointer.
INC_(PA) @	pointer reference with increment.
DEC_(PA) @	pointer reference with decrement.
3	reference to a constant, stored in the code.
35+R	constant + register contents.
XA @	
XA[IXA] @	
XA[INC_(IXA)] @	
XA[DEC_(IXA)] @	
PXA @ @	
INC_(PXA) @ @	
DEC_(PXA) @ @	

Figure Ap2-5: Addressable COL Expressions

-----

accessed in the address part of a single SUE instruction. More importantly, as long as such expressions are used in the statement forms shown in the next section, single SUE instructions will be produced.

## Ap2.1.4: SUE Instructions

Most machine instructions are expressed as normal COL expressions involving the above mentioned expressions as operands. We again proceed by example. Let S (for source) be any of the above expressions, and D (for destination) be any except the two which include constants, providing that either S or D (or both) designates a register. Then any COL statement of the following form is guaranteed to compile into a single SUE instruction:

```
D := S
D *= + S      // S, D integers
D *= - S      // S, D integers
D *= or S     // S, D logical
D *= and S    // S, D logical
D *= xor S    // S, D logical
```

Shift instructions shift in a register and take the shift count from the instruction or from a register. Denote this shift count by K, an integer, and let R be any expression denoting a register. Then each of the following statements compiles into a single SUE instruction:

```
R *= rshift K // R integer or logical
R *= lshift K // R integer or logical
R *= lrotate K // R logical
R *= rrotate K // R logical
```

SUE has two additional shift instructions. These rotate left and right, but include the carry bit in the shift. They are accessed from COL by

```
SLAO_(R, K) // R logical
SRAO_(R, K) // R logical
```



Branching in SUE is specified by a set of instructions which branch conditionally on the bits of the status register. These bits are declared as individual registers in COL (as described above), and a declaration for them appears on page 274. Each of these bits can be tested with a branch instruction. Let L in the following discussion be any expression of the form given above under addressing modes whose value is a statement label. Then each of

```
unless GT_ do goto L endunless
if OV_ do goto L endif
```

compiles into a single(43) SUE instruction. SUE has two unconditional branches, a one-word instruction if the destination is within 128 words and a two-word instruction for any branch. The COL statement

```
goto L
```

compiles into the better of the two; the programmer may use

```
BRANCH_(L)
```

to generate the long form. Two final branch instructions test the ZE\_ and GT\_ bits at the same time to test for less than. We express these

```
if GT_ and ZE_ do goto L endif
if (not GT_) and (not ZE_) do goto L endif
```

The use of this notation for testing ZE\_, EQ\_, LP\_, OD\_, ZE\_, and

---

(43) The label L must be within 128 words of this command for a one-word branch to be compiled. If L is further away the compiler compiles the correct multi-word instruction.



NG\_ is usually unnecessary since the notation is machine dependent, less compact and no faster than the normal COL notation. For example, while one could write

```
CMP_(A, B); if EQ_ do goto L endif
```

the same code is generated by

```
if A = B do goto L endif
```

which is shorter and not machine dependent. On the other hand,

```
X := A+B; if OV_ do goto ERROR endif
```

is the right way to test for overflow in adding A to B.

The remaining SUE instructions are obtained by machine-dependent subroutines. The compare instruction (already alluded to) and the test instruction serve to set bits in the status register, dependent on comparison of two values. COL declarations for CMP\_ and TST\_ are presented in Figure Ap2-6 to define their semantics in terms of the COL, but the reader should understand that uses of these routines compile into a single SUE instruction and not this code. The remaining SUE instructions are more specialized and are not presented in detail here, since the present purpose is to indicate the technique rather than to explain the SUE.

```

routine CMP_(A: integer, B: integer)
    EQ_ := A = B
    GR_ := A > B
endroutine

routine TST_(A: logical, B: logical)
    NG_ := force(A and B: integer) < 0
    ZE_ := (A and B) = 2#0
    OD_ := (A and B and 2#1) = 2#1
endroutine

```

Figure Ap2-6: COL Code for CMP\_ and TST\_

-----

```

routine JSBR_(ref R: register, L: label)
    // Subroutine jump to L, return address in R.
    R := PC_ // Save return address.
    goto L
endroutine

routine HALT_()
    // Halt the computer.
endroutine

routine RSTS_(A: logical) // Reset status.
    STATUS_ := STATUS_ and not (A and 8#177)
endroutine

routine SETS_(A: logical) // Set status.
    STATUS_ := STATUS_ or (A and 8#177)
endroutine

routine ENBL_(A: logical) // Enable interrupts.
    STATUS_ := STATUS_ and not (A lshift 12)
endroutine

routine ENBW_(A: logical) // Enable, wait.
    STATUS_ := STATUS_ and not (A lshift 12)
    WAIT_()
endroutine

```

```

routine DSBL_(A: logical)  // Disable interrupts.
    STATUS_ := STATUS_ or (A lshift 12)
endroutine

routine DSBW_(A: logical)  // Disable, wait.
    STATUS_ := STATUS_ or (A lshift 12)
    WAIT_()
endroutine

routine WAIT_()
    // Stop the computer till the next interrupt.
endroutine

// REGS is a type for a place in memory to copy registers.
declare (REGS is array[1..7] of logical)

routine REGM_(ref P: REGS)  // Registers to memory.
    // Store registers 1 to 7 into P[1] to P[7]
endroutine

routine MREG_(P: REGS)  // Memory to registers.
    // Load registers 1 to 7 from P[1] to P[7]
endroutine

routine RETN_()  // Return from an interrupt.
    // Return from an interrupt.
endroutine

```

#### Ap2.1.5: Events in SUE

In section 2.3 a mechanism is described that gives the programmer access to hardware-detected events. It is pointed out there that the actual events that can thus be detected are dependent on the particular implementation. For the SUE, the relevant events are overflow, carry and loop. These correspond respectively to bits 11, 12 and 16 of the status register. (See Figure Ap2-3 on page 276.) The identifiers OVERFLOW\_, CARRY\_ and LOOP\_ may be used to access these events. For example, within the scope of

```

    check LOOP_

```

every SUE instruction that includes auto-increment or auto-decrement of a register (i.e., each use of INC\_ or DEC\_) is followed by SUE instructions with effect equivalent to the COL statement

```
if LP_ do LOOP_ := true endif
```

The programmer clearly must use event declarations with caution.

#### Ap2.1.6: Interrupts and Input/Output

Both input-output and interrupts are controlled by manipulation of fixed memory locations. Interrupt control is through groups of four words each at the beginning (low addresses) of memory. The type INTDATA\_ describes such a block:

```
declare
( INTDATA_ is structure
  ( DA: address // interrupting device
    OST: logical // STATUS_ at time of interrupt
    OPC: address // program counter at interrupt
    NPC: address // location of interrupt processor
  )
)
```

Using this type we declare data blocks for four levels of interrupts:

```
declare
( LEVEL1_: location(16!0000) INTDATA_
  LEVEL2_: location(16!0008) INTDATA_
  LEVEL3_: location(16!0010) INTDATA_
  LEVEL4_: location(16!0018) INTDATA_
)
```

The effect of an interrupt from a device whose control registers are at address A is as if the following code were executed:



```

LEVEL1_.OST := STATUS_ // Save status register.
LEVEL1_.DA  := A       // Ident of interrupting device.
STATUS_     := 16#F000 // Set status.
LEVEL1_.OPC := PC_     // Save program counter.
goto LEVEL1_.NPC      // Off to do the work...

```

Of course, these five statements are executed as a single uninterruptible atomic act. Competing interrupts are resolved in favor of the highest level (greatest interrupt address). Any of the four levels of interrupts can be disabled by setting one of the left four bits of the status register.

Input/output is accomplished in the SUE by manipulating memory locations associated permanently with hardware devices. The standard arrangement for these addresses is illustrated by a teletypewriter input (keyboard) interface. Memory locations F800 to F807 (hex) are used to control such a device; a structure declaration describing this memory layout is shown in Figure Ap2-7. Given this declaration, the simplest input sequence is the following:

```

TTY.CONTROL := 2#0           // Clear all CONTROL bits
TTY.CONTROL.START := true    // Start it
until TTY.STATUS.PDT do until // Wait for data ready
X := TTY.DATA                // The character

```

A more complex example is the following program for teletype input using interrupts. We assume the above declaration for the teletype registers TTY as well as the earlier declaration of L1\_ for the interrupt vector. We assume that all level 1 interrupt routines lock out further interrupts at level 1, but allow interrupts at other levels. The code segments in Figure Ap2-8



```

declare
( TTY: location(16!F800) packed packed structure
  ( STATUS: structure
    ( : logical(12) // left 12 bits unused
      FMERR: boolean // no stop bit error
      OVRUN: boolean // data lost
      READY: boolean // true means teletype ready
      PDT: boolean // true means data ready
    )
    ADDRESS: integer // block transfer address
    LENGTH: integer // # words in block transfer
    CONTROL: structure
      ( : logical(10) // 10 bits unused
        MODEM: boolean // ask modem to send
        READ: boolean // reader in use
        ECHO: boolean // echo keyboard to printer
        INT: boolean // enable interrupts
        IO: boolean // true output, false input
        START: boolean // start the next op
      )
    DATA: logical // data goes here
  )
)

```

Figure Ap2-7: Teletype Control Block

-----

perform the functions indicated by the comments. If the actual processing of a character of input is fast enough so that it can be done while level 1 interrupts are disabled, the code for TTYINT could be

```

routine TTYINT()
  INPUT(TTY.DATA)
endroutine

```

(Here INPUT does the work.) If INPUT takes too much time, the code in Figure Ap2-9 might be used.

```

// This code is to process teletype interrupts. The
// following two statements are executed as part of system
// initialization:

// Arrange that level 1 interrupts be processed by LEV1SVC
LEVEL1_.NPC := LEV1SVC

// Initialize the teletype.
TTY.CONTROL := 2#001101           // echo, interrupt, input, go

// The following code does the work. It is placed outside
// the normal flow of control.

LEV1SVC:
  code SUE
    declare ( RSAVE: REGS )      // Place to save registers.
    ENBL (2#1110)                // Enable higher interrupts.
    REGM_ (RSAVE)                // Save all registers.
    switchon LEVEL1_.DA into      // Which device?
      case 15!BF00:              // Teletype.
        TTYINT()                // Do the work.
        stopswitch
      // Other cases would go here...
      default:                   // Hardware error?
        Error(1)
    endswitch
    MREG (RSAVE)                 // Restore the registers.
    RETN_ (LEVEL1_.OST)           // Return.
    data here (RSAVE)             // The place for the data.
  endcode

```

Figure Ap2-8: Code for Teletype Interrupt

-----

## Ap2.1.7: Unsigned Integer Multiply

As a further example of COL programming on SUE we present the algorithm for unsigned integer multiply found on page 8.9 of the SUE manual. This program is machine independent except for the

```

routine TTYINT()
  code SUE
    declare (TEMP: char )      // Temp to hold a character.
    TEMP := TTY.DATA
    STACK(LEVEL1 )             // Save LEVEL1 data
    ENBL (2#1000)               // Enable level 1
    INPUT(TEMP)                 // Do the work
    DSBL (2#1000)               // Disable level 1
    UNSTACK(LEVEL1_)            // Restore data
  endcode
endroutine

```

Figure Ap2-9: Teletype Interrupt Code

-----

width of the word. It is shown in Figure Ap2-10. The intent is

-----

```

function IMP(A: integer, B: integer): array[0..1] of integer
  code SUE
    declare ( X, I: register integer )
    X := 0
    for I in [0..15] do
      if (B and 2#1) = 2#1 do X := + A endif
      B := (B rshift 1) or (X lshift 15)
      X := rshift 1
    endfor
    result is construct(array[0..1] of integer, 0: X, 1: B)
  endcode
endfunction

```

Figure Ap2-10: Integer Multiply

that this should compile into the code given in the SUE manual. Although there are some problems in the details of meeting this objective, it is nonetheless an instructive example of SUE coding in COL.

#### Ap2.2: Parallel Processing and Co-Routines

COL contains no explicit language functions for either parallel processing or co-routines. As discussed in section 1.1.2, these functions can be implemented in terms of existing COL features. For example, parallel processing can be implemented by subroutines, and then co-routines can be implemented using parallel processing. In this appendix it is shown that this approach is both feasible and efficient. Of necessity, the routines needed for this demonstration are machine and implementation dependent; they are therefore given in terms of the SUE implementation just discussed.

##### Ap2.2.1: Functions for Parallel Processing

Two functions implement the capability for starting and coalescing processes.

V := START(PROG)

starts a new process in the form of a function call by executing the line

PROG()

PROG must be the name of a routine. The value V returned by START is an integer identifying the process. The process continues

running until PROG returns. The parent process can wait for the child to complete by executing

```
JOIN(V)
```

Communication between the two processes can be through static variables shared by the two functions or through variables declared external to both functions. All variables used in the routine PROG which are not used for communication should be declared locally so that they cannot be altered or used by the parent.

Communication and synchronization among processes is controlled through the use of interlocks. For example, the transmission of data from one process to another can be achieved simply through the use of a structure CHANNEL consisting of two interlocks and some room for data, like this:

```
declare
( CHANNEL is structure
  ( OK_TO_WRITE: interlock
    OK_TO_READ: interlock
    DATA: integer
  )
)
```

A static variable of this type is then accessed by both routines. Routines to send data on the channel and to receive data from it are shown in Figure Ap2-11. Note that this code uses the primitive statements "lock" and "unlock", rather than the structured "region" statement. (See the discussion in section 3.4.3.) The intent here is to exhibit a simple example of one way to implement co-routines, and it happens that the structured



```
routine SEND(ref V: CHANNEL, D: integer)
    lock V.OK_TO_WRITE
    V.DATA := D
    unlock V.OK_TO_READ
endroutine

function GET(ref V: CHANNEL): integer
    declare ( D: integer ) // A temp.
    lock V.OK_TO_READ
    D := V.DATA
    unlock V.OK_TO_WRITE
    result is D
endfunction
```

Figure Ap2-11: SEND and GET Procedures

-----

approach is not convenient. In these functions, the interlock OK\_TO\_WRITE is used to prevent the sender from overwriting something sent previously before it has been read, and OK\_TO\_READ is used to prevent the receiver from reading data before it has been put there by the sender.

#### Ap2.2.2: Implementation of START and JOIN

Many implementations for START and JOIN are possible, ranging from something as complex as a general scheduler for an operating system, to something as simple as co-routines. In communications programming, one is usually more interested in fast switching between processes than in elaborate structures for starting and stopping processes. In many systems the process structure is even fixed, in the sense that it is set up when the system is

initialized and not altered thereafter. Also switching between tasks is often voluntary, rather than being driven by a clock interrupt.

Recalling that we are starting with the SUE implementation, we make the following additional assumptions:

- . Routine or function calls are made using the subroutine jump instruction with register 7. Parameter addresses are stored after the jump instruction. The copying required for call by value is done in the routine, not by the caller.
- . The calling program saves its registers.
- . Automatic variables are stored on a stack, and the stack pointer is always in register 6.
- . The COL lock operation executes a call on routine SLEEP() if it finds the interlock to be set, then re-tests the interlock when SLEEP returns.

Switching between processes is initiated by a call on SLEEP. This call may be either explicit or a part of a lock operation. An interrupt may put the interrupted process to SLEEP too, but additional protocol is necessary for that; this problem is discussed separately below.

The variables used in the example are declared in Figure Ap2-12, and the code for the START function is in Figure Ap2-13.

```

-----
declare
( N:          // current number of processes [0..N]
  static integer
  M = 10      // maximum process number [0..M]
  PC:         // program counters
    static array[0..M] of address
  SP:         // stack pointers
    static array[0..M] of address
  C:          // index of the current process
    static integer
  STACK:      // the stacks
    static array[0..M] of array [0..500] of logical
)

```

Figure Ap2-12: Variables for Process Switching

```

-----

```

The first part of the program (up to GO) sets up the new process at the end of the array of processes, starting it off at location GO (inside of this function). When the SLEEP function comes to the new process, it jumps to GO, where the new stack pointer is initialized, and the subroutine call is initiated. When the function returns, the process loops in a call on SLEEP. This effectively takes it out of action; it is not removed from the list of processes until someone JOINS it. Note that the label

```

function START(PG: routine()): integer
  code
    if N >= M do fail endif  // too many processes
    N := N + 1
    PC[N] := GO
    SP[N] := PG  // temporarily save name of function
    result is N
  GO:
    R6 := ref (STACK[C][0])
    SP[C]()
    while true do SLEEP(); DEAD: endwhile
  endcode
endfunction

```

Figure Ap2-13: Code for START

-----

DEAD appears just after the call to SLEEP. The assumption is made in coding JOIN that the return address stored in register 7 when SLEEP is called is DEAD. Both SLEEP and JOIN are within the scope of an external declaration for DEAD.

The SLEEP routine changes processes when called. For simplicity we select processes in an untimed round robin fashion, so that the switch from one process to another requires only exchanging the program counters and stack pointers. The other registers have been saved by the function-call mechanism.

```

routine SLEEP()
  SP[C] := R6 ; PC[C] := R7
  C := when C >= N then 0 else C+1
  R6 := SP[C]; R7 := PC[C]
endroutine

```

The JOIN routine waits until the named process finishes, detecting



this condition by comparing its program counter to the address of the label DEAD in the function START at which processes terminate. JCIN then removes the awaited process from the list of processes.

```
routine JOIN(P: integer)
    while PC[P] ne DEAD do SLEEP() endwhile
    while N > 0 and PC[N-1] eq DEAD do N := N - 1 endwhile
endroutine
```

We now discuss the effectiveness of using these simple programs to implement co-routines and in interrupt processing.

#### Ap2.2.3: Co-routines

Co-routines can conveniently be viewed as parallel processes, with co-routine jump as a synchronization of two parallel processes in which some data is passed from one process to another(44). The routines SENDER and GETTER are started in parallel, and use the functions GET and SEND described above to exchange data. SENDER is viewed as a prototype data generator (lexical analysis in a compiler is the common example), and GETTER is the consumer (parsing in a compiler). The relevant code is in Figure Ap2-14. The co-routine jumps are in effect carried out by the call on SEND in SENDER and the call on GET in GETTER. The actual flow of control of these two programs is the following, assuming that only one processor is available.

1. Routine GETTER is started by the call on START but is not

---

(44) Passing data is not necessary but is usually done in conjunction with co-routines.



```

declare
(  CH: static CHANNEL    // The communication channel.
  PID: static integer    // Process ID for GETTER.
)

routine SENDER()
  // produce some data in D
  SEND(CH, D)
  // more computation, probably loop back
endroutine

routine GETTER()
  // initialize and start a loop
  D1 := GET(CH)
  // use the data from sender and loop back
endroutine

MAIN_PROGRAM:

CH.OK_TO_WRITE := unlocked
CH.OK_TO_READ  := locked
PID := START(GETTER)
SENDER()
JOIN(PID)

```

Figure Ap2-14: SENDER and GETTER for Co-routines

-----

activated until the first call on SLEEP

2. SENDER is called by the MAIN\_PROGRAM and continues execution until it has produced data in variable D.
3. In the first call on SEND(CH, D) the interlock CH.OK\_TO\_WRITE is not locked, so D is put into CH.DATA.
4. The interlock CH.OK\_TO\_READ is unlocked (it was locked initially). SENDER continues to run, loops back and comes to SEND again. At this point, CH.OK\_TO\_WRITE is still locked

from the last call on SEND, so a call on SLEEP results.

5. Sleep switches to the only other process on the list, namely GETTER. GETTER starts at its beginning, initializes, and comes to a call on GET. The interlock CH.OK\_TO\_READ is unlocked (there is data in CH.DATA) so it is locked, D1 := CH.DATA is done, and CH.OK\_TO\_WRITE is unlocked.
6. GETTER continues to execute, using up D1 and looping back eventually to make another call on GET. This time CH.OK\_TO\_READ is locked, so GET calls SLEEP, and SENDER is activated again.
7. SENDER retests the interlock (as a part of the lock operation) to make sure it is OK to go on. It is, since GETTER has unlocked CH.OK\_TO\_WRITE, so SENDER locks CH.OK\_TO\_WRITE, puts the second datum in CH.DATA and unlocks CH.OK\_TO\_READ.
8. SENDER continues to run until it has produced the third datum, then switches back to GETTER to consume the second, and so forth.
9. Control passes back and forth between the programs until one of them stops and returns. If SENDER stops first, the JOIN program is called (from MAIN\_PROGRAM), and since GETTER is still going (i.e., its PC is not DEAD), a call on SLEEP results activating GETTER again. GETTER finishes up (it had better do so without another call on GET or the program deadlocks) and returns. GETTER then goes into a permanent loop calling SLEEP, but SENDER detects this in JOIN, removes

GETTER from the process list, and the program ends.

10. If GETTER stops first, then it enters the permanent loop calling SLEEP. This enables SENDER to finish (again, it had better do so without calling SEND) and when JOIN is called, GETTER is already dead, so the program ends.

The main thing we are interested in here is the efficiency of switching between processes. To make this as fast as possible within the framework described, we declare SEND, GET and SLEEP as open routines to avoid the subroutine calling overhead. Once the steady state has been reached between the two programs, control always switches from one to the other at the beginning of the GET or SEND operation. SENDER has the following open code for implementing SEND(CH, D):

1. Lock CH.OK\_TO\_WRITE. This always fails in the steady state, so we go on to the next step.
2. The code in SLEEP switches processes.
3. On return from SLEEP, retest the lock. This always succeeds.
4. Unlock CH.OK\_TO\_READ.

The code in GETTER is essentially the same, reversing the roles of CH.OK\_TO\_SEND and CH.OK\_TO\_READ. Inasmuch as the operations of item 2 just above and any other saving of registers are an essential part of switching from one program to another, any co-routine implementation would have to do the equivalent operations. The extra overhead from the parallel processing environment is the testing and clearing of the interlocks.

The advantage in tailoring things this way, rather than just writing some special code for these particular co-routines (which is clearly possible), is that the resulting program is more readable. The whole structure is understood in terms of the more general parallel processing primitives. The comments about the special implementation warn the reader to be wary of changing the process setup while these routines are running, lest the special implementation become invalid.

#### Ap2.2.4: Interrupt Handling

An interrupt program is always a process running in parallel with other interrupt programs and with processes, in the sense of the preceding paragraphs. Thus, setting up the interrupt vectors and enabling the interrupt is functionally equivalent to calling

START(INTPG)

where INTPG is the name of the interrupt program. The difference between processes in the conventional sense and interrupt programs is entirely in the method used for switching between processes. In the above programs the switch is initiated by a call on SLEEP; with interrupt programs, it is initiated by the hardware interrupt action. In the latter case only the program counter and status register are exchanged. The stack pointer, other registers, and the variable C (which designates the active process) are not switched. We could completely merge the two systems for parallel processing by providing a place in our declarations for the other registers (1-5 and the status register) and then code the entrance



sequence for interrupt programs to simulate in effect a call on SLEEP on behalf of the interrupted program. In this case all interrupts would be viewed as a process switch in the sense of our programs given above, and the return from interrupt would be a special sort of call on SLEEP (the interrupt program goes to SLEEP until the next interrupt occurs). We would need a few alterations in our programs to deal with SLEEPing programs and the priority of interrupts, but they would not be fundamentally more complicated.

Unfortunately, this way of viewing the world forces a significant overhead on by far the most common actual situation in an interrupt, namely, that a little simple processing is done (typically storing away the next datum from a device) and then the interrupted process resumes with a simple return from interrupt. A good compromise is to allow the interrupt program to choose. In most situations the interrupt program does not inform the process management programs of the interrupt, but only takes its brief action (like transmitting data to or from a device) which does not affect other processes and returns to the interrupted program. Only in cases where some interaction with another process takes place (for example, enough data has been gathered to enable it to be worked on by another process) does process management intervene. In these cases, the usual action is to unlock an interlock as a signal to another process, and then to suggest process switching. The best implementation for this "suggesting" function is a special sort of SLEEP routine for interrupt programs



which is called when the interrupt program has finished its business. The action of the function is to return from interrupt, but not to go back to the interrupted program. Instead the process switch is invoked to effect a forced call of SLEEP into the interrupted program so as to allow other processes to get their chance.

### Appendix III: Terminology Used in this Document

Some of the terminology used in this document is not completely standard in the profession. So as to remove possible doubt about meaning, all such terms are defined in this appendix and used consistently with that meaning.

actual parameter  
See parameter.

address, addressable  
See memory.

argument  
The terms ARGUMENT and parameter are used more or less interchangeably in the literature to refer to the argument or parameter of a procedure. Only "parameter" is used in this document; see parameter.

assertion  
An ASSERTION is a claim made by the programmer and included in his program, with syntax described in section 6.4. Using a suitable compiler directive (see section 5.3.2), the programmer may direct what action is to be taken for each assertion. The presence of assertions can help in debugging and in verification.

block  
A BLOCK in the COL is any context in which declarations may appear intermixed with statements. Syntactically, it is any place at which "SD ; ..." is used in the syntax. A complete list of such places is given in section 2.7 on page 82.

bound  
See scope.

call type  
This refers to the method used to bind an actual parameter to a formal parameter. The default in the COL is no call type

is specified is READ ONLY, which means that the value of the actual parameter may be accessed but that neither the actual parameter nor the formal parameter may be updated. With CALL BY VALUE the R-value of the actual parameter is calculated and used to initialize the cell in the procedure body corresponding to the formal parameter. An update of the formal parameter cannot affect the calling point. With CALL BY REFERENCE the L-value of the actual parameter is bound to the formal parameter, so that an update of the formal parameter within the procedure body changes a cell in the calling point. The ALGOL-60 concept of call by name is not available in the COL.

capsule

See encapsulated data type.

cell

See memory.

class

The language fragment defined by a single BNF definition is called a CLASS. All of the COL's classes are listed in the first part of Appendix I.

closed

A procedure is said to be CLOSED if there is only one copy of the procedure's code in the computer, a subroutine-calling sequence being used to activate that copy from multiple places in the program. Compare with open.

coercion

A COERCION is a change in type performed by the compiler when the programmer uses an expression of one type in a context where another type is required. There are very few coercions in the COL; they are described in section 6.1.5.

conformable

Two aggregates are said to be CONFORMABLE if they have the same type. That is, arrays have the same number of dimensions and the same limits in each dimension, and structures must have the same field names and the same types in each field.

constant

See identifier.

convert

A datum with a given type may be CONVERTed to another type, using a process that preserves (in some sense) the value while changing its representation.

data type  
See type.

dynamic  
See storage class.

encapsulated data type  
Syntactically, an encapsulated data type or CAPSULE is a structure in which other declarations appear. Because the programmer can make some or all of the capsule PRIVATE and so not known outside the capsule, he has the power to hide certain aspects of an implementation from all code outside the capsule. Only those parts marked PUBLIC are available. Further, part of the capsule may define FINALIZATION, code to be executed each time a variable of that type is de-allocated. Capsules are described in section 6.1.3.

error  
The term "compiler error" refers to an error made by the user of the language which is detected during the compilation process by the compiler. Other user errors are detected by the linker or (rarely) at run time. The distinction is always made clear in the text.

event  
An EVENT is a hardware-detected phenomenon such as arithmetic overflow which the programmer may want to be warned about. The event declaration  
    check overflow  
may be used to force the necessary checks, as described in sections 2.3 and 6.1.4.1.

expression  
An EXPRESSION is a fragment of COL text that is evaluated to determine its value. Usually the evaluation of an expression has no side effects. Compare with the definition of statement.

finalization  
See encapsulated data type.

flexible  
An array is said to be FLEXIBLE if either or both of its bound limits are unknown at compile time.

force  
Every expression has both a data type, determined at compile time, and a value, which is usually not known until run time. The FORCE mechanism in the COL permits the programmer to direct the compiler to interpret the data as having some



different type. See section 4.4.

formal parameter  
See parameter.

free  
For discussion of the concept of a variable being free at a certain point, see scope. The word is also used in connection with the "free" statement to return space to free storage.

function  
See procedure.

govern  
See scope.

identifier  
An IDENTIFIER is the name used in the source text of a COL program to denote some value. Every identifier used must be within the scope of a declaration that tells the compiler certain things about the identifier and about the place where its value is stored. A VARIABLE is an identifier whose value changes (or at least might well change) during the execution of the program. A CONSTANT is an identifier whose value is known at compile time. A LITERAL is a special name such that the value which it denotes is immediately deducible from the form of the name; examples are

12    \$A    true    "string"

To see the distinction between these, consider the following COL declaration:

declare (VAR: integer; CON = 4)

Two identifiers appear: the variable VAR declared to be of type integer, and the constant CON which throughout its scope always has the value four. There is also the literal 4 whose value is obvious from its form.

label constant  
A LABEL CONSTANT is an identifier which labels a place in the program. Syntactically it is an ID followed by a colon appearing before a statement. It is "constant" in the sense that the value of the ID is known at compile time and cannot be changed at run time.

lexeme  
A LEXEME is a single lexical token in the language, such as an identifier or a literal or a reserved word or a punctuation such as one of the following:

( + := <= .. : [ {

A lexeme may not contain embedded space.



**link**

The process of preparing a large program for execution on a computer involves first compiling all of its parts and then binding those parts together. The latter is referred to as LINKING, and is done by a LINKER. The term "link" is always used in this way in this document.

**literal**

See identifier.

**L-mode, L-value**

See memory.

**memory**

The COL design assumes that programs run in a computer with a MEMORY which is organized into units called WORDs, each of which has an ADDRESS. A BYTE is that unit of memory such that the address of successive bytes differs by one. (There may be more than one byte per word.) A CELL is a place in the memory that can hold a value. The term L-VALUE corresponds to the address of a cell, and the term R-VALUE to the contents of a cell. The letters L and R are mnemonic for Left and Right, respectively, since in an assignment statement such as

$$X := X + 1$$

X's L-value is needed on the left side and its R-value on the right side. Note that every expression has an R-value but that some expressions do not have an L-value. A constant does not have an L-value nor does a calculated expression such as  $X+1$ .

Every item in the memory is said to have an L-value, but it is not the case that every such item is ADDRESSABLE. The latter means that the item's boundaries correspond to byte boundaries. A component of a packed structure has an L-value (and so may appear on the left side of an assignment statement), but it is not necessarily addressable and so may not be an actual parameter called by reference.

**module**

Although this term means various things in the literature, it is used in this document to mean a piece of code which can be compiled separately. See section 6.2.

**name, call by**

See call type.

**open**

A procedure is said to be OPEN if the code for the procedure appears as many places as there are invocations of the

procedure. The binding of actual parameters to formal parameters is accomplished by substitution, with care so that the semantics remains unchanged if the mode is changed to closed. Compare with closed.

operand, operator

An OPERATOR is a mark in the language that denotes a function on one or two arguments, called OPERANDS. Most of COL's operators are infix, such as "+", "/", "and", etc. Both "+" and "-" are prefix operators, and "@" may be thought of as a postfix operator.

parameter

The terms PARAMETER and ARGUMENT are used somewhat interchangeably in the literature for two purposes. In this document we follow ALGOL-60 and use the term FORMAL PARAMETER to refer to the identifier used in a procedure declaration to stand for the parameter, and the term ACTUAL PARAMETER for the expression that appears in the invocation of the procedure.

polymorphic

An operator or procedure is said to be POLYMORPHIC if it is prepared to accept arguments of differing types. For example, the arithmetic operators in the COL (as well as in most high order languages) are polymorphic, being prepared to operate on two integers or two floating point numbers. The COL does not provide a mechanism for the user to write a polymorphic procedure.

portable

A piece of code is said to be PORTABLE if it can be carried fairly readily from the machine for which it was written to another machine.

private

See encapsulated data type.

procedure

A procedure is a subroutine that can be invoked from other parts of a program, the invocation usually involving the passing of actual parameters at the invocation that are bound to formal parameters listed in the definition. A ROUTINE is a procedure obeyed for its effect; an invocation of a routine may appear wherever a statement may appear. A FUNCTION is a procedure that calculates a value which it returns; it may be used in any expression context.

public

See encapsulated data type.

read only, call by

See call type.

re-entrant procedure

A procedure is said to be RE-ENTRANT if it can be re-entered at the top at a time when it is currently being obeyed. A recursive procedure is re-entrant, but so also is a procedure that may be obeyed as part of interrupt processing while it is being obeyed for some other (or the same) reason. All procedures are assumed to be potentially re-entrant by the COL compiler.

reference, call by

See call type.

reserved word

A reserved word is a lexeme that is lexicographically indistinguishable from an identifier but which is reserved for use by the language, so that the user may not have an identifier spelled the same. The COL uses reserved words. (Contrast PL/I which has no reserved words, so that the programmer may have identifiers such as "declare" or "if", etc.)

routine

See procedure.

R-mode, R-value

See memory.

scope

The COL is a fully typed language, which means that every variable used in a program must be declared. For any USING INSTANCE of a variable, it is necessary to be able to determine which declaration is the DEFINING INSTANCE that gives the type information for that use. The SCOPE of a declaration is that part of the program text in which using instances are bound to that declaration. Note that it is a declaration that has a scope -- not an identifier. A using instance of an identifier is said to be BOUND in a piece of text if there is in that text a defining instance which the use is within the scope of. A using instance is FREE in a piece of text if it is not bound within that text. The COL's scope rules are presented in section 2.7.

**side effect**

Normally evaluation of an expression has no effect other than to return the expression's value, but in some cases some item may be changed as a consequence of the evaluation. Such a change is a SIDE EFFECT of the evaluation. In the COL, side effects may occur in only two ways: A function called as part of the evaluation may update a global variable, and an "event" variable may be set. It is good programming practice to minimize the use of side effects.

**statement**

A STATEMENT is a fragment of COL text which is obeyed for the effect it has; it does not have a value. See also expression.

**static**

See storage class.

**storage class**

Each variable is assigned a cell in memory, either before or during execution of the program. A STATIC variable is assigned a cell before execution of the program and retains that cell throughout the lifetime of the program. A value stored into that variable will be intact at any time in the future. A DYNAMIC variable is assigned a cell at the time its scope is entered while the program is running, and nothing may be assumed about its initial value.

**syntactic sugaring**

This refers to a syntactic device which, while giving the user nothing that he did not already have, nonetheless permits him to express something in a more convenient manner. For example, arithmetic operations could in principle be expressed functionally, as

PLUS(A, TIMES(B, C))

The infix notation  $A+B*C$  is a clearly desirable syntactic sugaring. The COL provides syntactic sugaring that permits an array of arrays to be declared and accessed as if it were a multiply subscripted array.

**type**

All possible values which a COL program may deal with are divided into mutually exclusive subsets, each of which is characterized by a descriptor called its TYPE. See the discussion at the beginning of Chapter 2.

**update**

The effect of an assignment statement is to UPDATE the item whose L-value appears on the left.



value, call by  
See call type.

variable  
See identifier.

variadic  
A procedure is said to be VARIADIC if the number of actual parameters may be different on different invocations.



## Appendix IV: Comparison with the "Ironman"

A matter of current interest in the Department of Defense is standardizing on a High Order Language (HOL) for DoD applications. The latest result of this effort is the so-called Ironman Report [DoD-77] which presents criteria which candidate languages are to meet. Although the COL was not intended to meet these criteria, it nonetheless seems useful to note those areas in which it fails to. There are few serious differences.

The remainder of this appendix lists briefly those points made in the Ironman with which the COL disagrees, keyed to the section numbers in that document. A given difference is mentioned only once. For example, having mentioned under 3A that the COL does not include fixed point numbers, we do not later comment on those Ironman requirements relating to fixed point.

It is pointed out in section 1.4 of this document that the Ironman was released too late in the design of the COL for it to have any impact on the language's design. A phrase such as, "We might change this," points out an area in which we might well change the COL to match the Ironman, had we sufficient time to consider the implications of the change.

2D        The COL notes the existence of source program line boundaries in three ways, as listed in section 5.5.1. Of these only the semicolon rule differs from the spirit of the Ironman, and in our opinion the difference is a small one. We like the semicolon rule. See also 2H and 2I.

2E        The COL's macro ability permits identifiers or key words to be abbreviated. We agree that doing so is bad style but cannot prohibit it without excising macros, a feature which we want.

2H        The COL includes a special convention to quote a string across line boundaries, but the programmer must indicate explicitly his intention to do so.

2I        One of the COL's comment conventions crosses line boundaries. We might change this.

3-1A      The COL does not admit fixed point numbers.

3-3E      The COL does not include catenation. The concept does not seem relevant to communication.

3-3G      The COL does not include non-updatable record components. This seems like a good idea.

3-3I      The COL does not include dynamic data types. We see no way to implement them efficiently.

3-4A     The COL does not include sets as a type. We see no way to implement them efficiently, and they seem to have little relevance to communications.

3-5D     It is not possible in the COL to define operations common to two capsules. It seems like a good idea but we do not see how to do it.

4A       Inasmuch as the COL provides no way to extend the definitions of built-in operators to programmer-defined types, operations on such types cannot use those operators and hence look different.

4C       The COL does not include this restriction about side effects. Although we agree that violating the restriction is bad coding practice in the extreme, we feel that the restriction cannot be enforced without imposing crippling limitations on the power of the language. See 7E.

4F       The COL has more than "three or four" precedence levels. (It has twelve of them.) We feel that a smaller number is less convenient for the programmer.

5A       The COL does not permit constants whose value is determined at scope entry time. This is a good idea which we would add.

5D The COL includes variables of type function and procedure.

The COL's scope rules regarding access within a procedure to non-local dynamic variables make it possible for the COL to implement them efficiently.

6D The COL requires that boolean expressions in conditional context be evaluated from left to right in short circuit mode. It is not clear whether or not we disagree with the Ironman.

6F The COL's "for defined" iteration statement uses a loop variable which is not local to the controlled body. We think that useful. Assignment is permitted to the loop variable. We would change this.

6G The COL permits transfer into conditional control structures. There is no ambiguity about the resulting flow of control.

7A It is not possible in the COL to extend built-in functions and operators to programmer-defined types. See 4A.

7E The COL does not include this restriction. It permits updating actual parameters called by reference, and it permits updates within a function body to variables whose scope includes the calling point. Our failure to meet 4C follows from our failure to meet this requirement.



7F        The COL does not have "output parameters". It does have call by read only, not mentioned in the Ironman. The output parameter seems like a useful concept.

7I        The COL design permits aliasing. This is the problem alluded to in 7E.

8A - 8C   The COL includes no input/output.

9A        The COL syntax includes no control structure for parallel operations. These must be programmed; that it is easy to do so is shown in Appendix II.

9E, 9F    No clock services are part of the COL design. Many mini-computers used in communications do not have a real time clock, and those that have one provide it in various different ways.

10B       The COL design does not permit detecting an access of an uninitialized variable at run time, although the compiler is able to detect this situation in many cases. We do not know how to implement such a run time check efficiently in most hardware architectures.

11B       All instances of a given type use the same packing strategy.

12A       The nature of libraries is not a language issue and is in any case beyond our control. However, it is our intention



that any implementation include libraries as described here.

12D      We do not understand this requirement but suspect that we do not meet it. The COL does not have parameterized types.

13        These are beyond our control but are in agreement with our intent.

## Appendix V: Proposed Changes in the COL

There are some aspects of the COL that we would change had we more time.

Extend the built-in operators to operate on defined types. One might write something like

```
function "+"(A, B: complex): complex
...
endfunction
```

to define addition on type complex.

Permit parameterized types, with all actual parameters supplied at compile time. For example, having declared

```
declare (ARRAY(N) is array[1..N] of integer)
```

one could later write

```
declare (P: ARRAY(5))
```

to declare P to be an array from one to five of integers. One can get almost the same effect with macros, but not quite.

It would frequently be helpful to be able to pass parameters to the initialization code in a capsule. The parameter would have to be supplied as part of the declaration of each instance.

There should be the ability to specify of a public variable that it is read-only, or perhaps that should be the default. A

separate open routine could be provided to update each such variable, but that routine might be public to fewer modules. This feature would provide effective control of what modules could modify global variables.

It should be possible to direct conditional execution of the compiler directive "%message".

In declaring a variable which is an aggregate and giving an initial value with the "construct" feature, the present syntax requires that the type be written out twice. A sugaring to eliminate this repetition would be useful.

A structure whose last element is a flexible array is frequently helpful in practice but is not possible in the present COL.

Although the value returned by a function may be an aggregate, there is no effective way to use this feature unless the aggregate can be a "construct" expression. This is because the result has no name in the function body.

A compile time "size" function to return the size of an aggregate (or part of an aggregate) is frequently helpful.

Further specification is needed of the effect of an assertion or check that fails. The programmer needs some way to get control and, presumably, to return to the program.

There should be read-only variables set only by the "initially" attribute. These are effectively dynamic constants, set on entrance to their scope. (This feature is required by the Ironman.)

The following points were suggested by our study of proof rules for the COL.

The "for defined" feature is questionable, as is the ability to update the controlled variable within the body of a for-statement.

Each procedure declaration should have a "sets" and "uses" list of global variables that it updates and accesses, respectively.

The ability to signal from the second part of a Zahn is probably not good. It is effectively a "goto" and should not be encouraged.

## Index

The following index lists the page on which various topics are referred to in the text. If a semicolon appears in the listing, the references before it are of greater importance than those after it. Otherwise the references are in the order of their appearance in the text.

abnormal exit	121, 123, 125, 209
allocate at link time	55; 28, 38 (footnote 4), 233
arguments, check type of	10, 27, 218, 224
assertion	section 6.4; 92, 172, 193, 201, 216
block	81; 39, 83, 125
call type	46, 215
coerce	section 6.1.5; 38, 94
comment conventions	section 5.5.2; 77 (footnote 11), 169
condition	(see "Zahn's device")
data declaration	27, 218
default	section 6.1.7; 25
different	57, 212
encapsulated type	67, 173, 210, 215
enumeration	76; 79
event	section 2.3; 203, section Ap2.1.5
exception handling	section 6.1.4; 7, section 2.3, section 3.4.2, section 3.4.4
failure	section 3.4.4; 82, 121, 208
flexible arrays	section 6.1.1.2; 77, section 6.1.1.1
force	25, 75, 156
free storage	12, 29, 111, 150
general	25, 75



include	171; 23
interlock	section 3.4.3; 133, 242
label	53, 56, 83, 84, 100, 110
linker	section 1.3.3; 38 (footnote 4), 172
machine code	section 6.3; 7, 13, 25, 75, 82, 92, 156
macro	section 2.6, section 5.2, section 6.1.2.3
module	23, 82, 217, 223
open code	section 6.1.2.2; 43, 215
packing	69
parallel	72
pointer	74, 132, 151, 156, 173
recursion	40, 42, 86
re-entrant code	14, 45
region	118, 125
relative binary	26, 27
scope	section 2.7; 46, 52, 110, 114, 116
semicolon insertion	section 5.5.3; 77 (footnote 11)
stack	14, 45, 51, 53, 84, 174
subrange	75; 78
variadic procedures	section 6.1.1.3; 48, section 6.1.1.1, 239
warning	26, 172
Zahn's device	section 3.4.2; 64, 206, 242